TMS320C55x Chip Support Library API Reference Guide

SPRU433
April 2001

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, license, warranty or endorsement thereof.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations and notices. Representation or reproduction of this information with alteration voids all warranties provided for an associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Resale of TI's products or services with *statements different from or beyond the parameters* stated by TI for that products or service voids all express and any implied warranties for the associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Also see: Standard Terms and Conditions of Sale for Semiconductor Products.
www.ti.com/sc/docs/stdterms.htm

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Preface

Read This First

About This Manual

The TMS320C55x™ DSP Chip Support Library (CSL) provides C-program functions to configure and control on-chip peripherals. It is intended to make it easier to get algorithms running in a real system. The goal is peripheral ease of use, shortened development time, portability, and hardware abstraction, with some level of standardization and compatibility among devices. A version of the CSL is available for all TMS320C55x™ DSP devices.

This document provides reference information for the CSL library and is organized as follows:

- Overview – high level overview of the CSL
- How to use CSL – Configuration and use of the DSP/BIOS™ Configuration Tool, installation, coding, compiling, linking, macros, etc.
- Using the DSP/BIOS™ Configuration Tool with the different CSL Modules
- Using CSL functions and macros with each individual CSL module.
- Using the individual registers.

How to Use This Manual

The information in this document describes the contents of the TMS320C5000™ DSP Chip Support Library (CSL) as follows:

- Chapter 1 provides an overview of the CSL, includes tables showing CSL API module support for various C5000 devices, and lists the API modules.
- Chapter 2 provides basic examples of how to use CSL functions with or without using the DSP/BIOS™ Configuration Tool, and shows how to define build options in the Code Composer Studio™ environment.

- Chapter 3 provides basic examples of how to configure the individual CSL modules using the DSP/BIOS™ Configuration Tool.
- Chapters 4-15 provide basic examples, functions, and macros for the individual CSL modules.
- Appendix A provides examples of how to use CSL C5000 Registers.

Notational Conventions

This document uses the following conventions:

- Program listings, program examples, and interactive displays are shown in a `special typeface`.
- In syntax descriptions, the function or macro appears in a **bold typeface** and the parameters appear in plainface within parentheses. Portions of a syntax that are in **bold** should be entered as shown; portions of a syntax that are within parentheses describe the type of information that should be entered.
- Macro names are written in uppercase text; function names are written in lowercase.
- TMS320C55x™ DSP devices are referred to throughout this reference guide as C5501, C5502, etc.

Related Documentation From Texas Instruments

The following books describe the TMS320C55x™ DSP and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477-8924. When ordering, please identify the book by its title and literature number. Many of these documents are located on the internet at http://www.ti.com.

TMS320C55x DSP Algebraic Instruction Set Reference Guide (literature number SPRU375) describes the algebraic instructions individually. Also includes a summary of the instruction set,a list of the instruction opcodes, and a cross-reference to the mnemonic instruction set.

TMS320C55x Assembly Language Tools User's Guide (literature number SPRU280) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for TMS320C55x devices.

TMS320C55x Optimizing C Compiler User's Guide (literature number SPRU281) describes the 'C55x C Compiler. This C compiler accepts ANSI standard C source code and produces assembly language source code for TMS320C55x devices.

TMS320C55x DSP CPU Reference Guide (literature number SPRU371) describes the architecture, registers, and operation of the CPU for these digital signal processors (DSPs). This book also describes how to make individual portions of the DSP inactive to save power.

TMS320C55x DSP Mnemonic Instruction Set Reference Guide (literature number SPRU374) describes the mnemonic instructions individually. Also includes a summary of the instruction set, a list of the instruction opcodes, and a cross-reference to the algebraic instruction set.

TMS320C55x Programmer's Guide (literature number SPRU376) describes ways to optimize C and assembly code for the TMS320C55x DSPs and explains how to write code that uses special features and instructions of the DSP.

TMS320C55x Technical Overview (SPRU393). This overview is an introduction to the TMS320C55x digital signal processor (DSP). The TMS320C55x is the latest generation of fixed-point DSPs in the TMS320C5000 DSP platform. Like the previous generations, this processor is optimized for high performance and low-power operation. This book describes the CPU architecture, low-power enhancements, and embedded emulation features of the TMS320C55x.

Trademarks

The Texas Instruments logo and Texas Instruments are registered trademarks of Texas Instruments. Trademarks of Texas Instruments include: TI, Code Composer, DSP/BIOS, and TMS320C5000.

All other brand or product names are trademarks or registered trademarks of their respective companies or organizations.

Contents

1 CSL Overview **1-1**
An overview of the features and architecture of the Chip Support Library
1.1 Introduction to CSL 1-2
1.2 Naming Conventions 1-5
1.3 Data Types 1-6
1.3.1 Resource Management 1-6
1.4 Symbolic Constant Values 1-8
1.5 Macros 1-9
1.6 Functions 1-11
1.6.1 Initializing Registers 1-12

2 How to Use CSL **2-1**
Detailed instructions and examples for the configuration of CSL DSP/BIOS
2.1 Installing the Chip Support Library 2-2
2.2 Overview 2-3
2.3 DSP/BIOS Configuration Tool: CSL Tree 2-4
2.4 Generation of the C Files (CSL APIs) 2-8
2.4.1 Header File projectcfg.h 2-8
2.4.2 Source File projectcfg_c.c 2-9
2.5 Modifying the Project Folder 2-12
2.5.1 Modification of C code (main.c) 2-14
2.6 Example of CSL APIs Generation (TIMER Module) 2-15
2.6.1 Configuration of the TIMER1 Device 2-15
2.6.2 Generation of C Files 2-17
2.7 Configuring Peripherals Without GUI 2-20
2.7.1 Using DMA_config() 2-20
2.7.2 Using DMA_configArgs() 2-22
2.8 Compiling and Linking With CSL 2-24
2.8.1 Using the DOS Command Line 2-24
2.8.2 Using the Code Composer Studio Project Environment 2-25
2.8.3 Creating a Linker Command File 2-30
2.9 Rebuilding CSL 2-31
2.10 Using Function Inlining 2-31

3 DSP/BIOS Configuration Tool: CSL Modules . . . 3-1
Detailed explanation for using specific modules when configuring CSL
3.1 Overview . . . 3-2
3.2 CACHE Module . . . 3-3
3.2.1 Overview . . . 3-3
3.2.2 CACHE Configuration Manager . . . 3-3
3.2.3 CACHE Resource Manager . . . 3-5
3.2.4 C Code Generation for CACHE Module . . . 3-6
3.3 DMA Module . . . 3-8
3.3.1 Overview . . . 3-8
3.3.2 DMA Configuration Manager . . . 3-8
3.3.3 DMA Resource Manager . . . 3-10
3.3.4 C Code Generation for DMA Module . . . 3-12
3.4 EMIF Module . . . 3-14
3.4.1 Overview . . . 3-14
3.4.2 EMIF Configuration Manager . . . 3-14
3.4.3 EMIF Resource Manager . . . 3-16
3.4.4 C Code Generation for EMIF Module . . . 3-18
3.5 GPIO Module . . . 3-20
3.5.1 Overview . . . 3-20
3.5.2 Non-Multiplexed GPIO Configuration Manager . . . 3-20
3.5.3 C Code Generation for GPIO Module . . . 3-21
3.6 MCBSP Module . . . 3-23
3.6.1 Overview . . . 3-23
3.6.2 MCBSP Configuration Manager . . . 3-23
3.6.3 MCBSP Resource Manager . . . 3-26
3.6.4 C Code Generation for MCBSP Module . . . 3-27
3.7 PLL Module . . . 3-30
3.7.1 Overview . . . 3-30
3.7.2 PLL Configuration Manager . . . 3-30
3.7.3 PLL Resource Manager . . . 3-32
3.7.4 C Code Generation for PLL Module . . . 3-33
3.8 TIMER Module . . . 3-35
3.8.1 Overview . . . 3-35
3.8.2 TIMER Configuration Manager . . . 3-35
3.8.3 TIMER Resource Manager . . . 3-37
3.8.4 C Code Generation for TIMER . . . 3-39

4 CACHE Module . . . 4-1
Descriptions and examples of the structure, functions, and macros contained in the CACHE Module
4.1 Overview . . . 4-2
4.2 Configuration Structure . . . 4-4
4.3 Functions . . . 4-5
4.4 Macros . . . 4-8

5 CHIP Module 5-1
Descriptions and examples of the functions used in the CHIP Module
5.1 Overview 5-2
5.2 Functions 5-3

6 DAT Module 6-1
Descriptions and example of the functions of the DAT module
6.1 Overview 6-2
6.2 Functions 6-3

7 DMA Module 7-1
A description of the structure, functions and macros of the DMA module
7.1 Overview 7-2
7.2 Configuration Structure 7-4
7.3 Functions 7-6
7.4 Macros 7-11
7.5 Examples 7-13

8 EMIF Module 8-1
A description of the structure, functions, and macros of the EMIF module
8.1 Overview 8-2
8.2 Configuration Structure 8-5
8.3 Functions 8-7
8.4 Macros 8-10

9 GPIO Module 9-1
General description of the macros available in the GPIO module
9.1 Overview 9-2
9.2 Macros 9-3

10 IRQ Module 10-1
General description of the IRQ module and its functions
10.1 Overview 10-2
10.2 Configuration Structure 10-7
10.3 Functions 10-8

11 McBSP Module 11-1
Description of the structure, functions, and macros of the MCBSP module
11.1 Overview 11-2
11.2 Configuration Structure 11-4
11.3 Functions 11-6
11.4 Macros 11-12
11.5 Examples 11-14

12 PLL Module 12-1

Describes the structure, functions, and macros of the PLL module

12.1 Overview 12-2
12.2 Configuration Structure 12-4
12.3 Functions 12-5
12.4 Macros 12-8

13 PWR Module 13-1

General description of the PWR module's functions and macros

13.1 Overview 13-2
13.2 Functions 13-3
13.3 Macros 13-4

14 TIMER Module 14-1

Description of the structure, functions, and macros of the TIMER module

14.1 Overview 14-2
14.2 Configuration Structure 14-4
14.3 Functions 14-5
14.4 Macros 14-9

Figures

1–1 API Modules 1-3
2–1 CSL Tree 2-4
2–2 Expanded CSL Tree 2-5
2–3 Insert Configuration Object 2-6
2–4 Delete/Rename Options 2-7
2–5 Show Dependency Option 2-7
2–6 Resource Manager Properties Page 2-10
2–7 Practice Summary 2-13
2–8 CCS Project View 2-16
2–9 Configuring the TIMER1 Device 2-16
2–10 Header File mytimercfg.h 2-17
2–11 Source File mytimercfg_c.c 2-18
2–12 Example of main.c File Using Data Generated by the Configuration Tool 2-19
2–13 Defining the Target Device in the Build Options Dialog 2-26
2–14 Defining Large Memory Model 2-27
2–15 Adding the Include Search Path 2-28
2–16 Defining Library Paths 2-29
3–1 CACHE Sections Menu 3-3
3–2 CACHE Properties Page 3-4
3–3 CACHE Resource Manager Menu 3-5
3–4 CACHE Properties Page With Handle Object Accessible 3-6
3–5 DMA Sections Menu 3-8
3–6 DMA Properties Page 3-10
3–7 DMA Resource Manager Menu 3-11
3–8 DMA Properties Page With Handle Object Accessible 3-12
3–9 EMIF Sections Menu 3-14
3–10 EMIF Properties Page 3-16
3–11 EMIF Resource Manager Dialog Box 3-17
3–12 GPIO Sections Menu 3-20
3–13 GPIO Properties Page 3-21
3–14 MCBSP Sections Menu 3-23
3–15 MCBSP Properties Page 3-25
3–16 MCBSP Resource Manager Menu 3-26
3–17 MCBSP Properties Page With Handle Object Accessible 3-27
3–18 PLL Sections Menu 3-30
3–19 PLL Properties Page 3-31

3–20 PLL Resource Manager Menu 3-32
3–21 PLL Properties Page 3-33
3–22 Timer Sections Menu 3-35
3–23 TIMER Properties Page 3-37
3–24 Timer Resource Manager Menu 3-37
3–25 Timer Properties Page With Handle Object Accessible 3-39
7–1 DMA Channel Initialization Using DMA_config() 7-13

Tables

1–1	CSL Modules and Include Files	1-3
1–2	CSL Device Support	1-4
1–3	CSL Naming Conventions	1-5
1–4	CSL Data Types	1-6
1–5	Generic CSL Symbolic Constants	1-8
1–6	Generic CSL Macros	1-9
1–7	Generic CSL Macros (Handle-based)	1-10
1–8	Generic CSL Functions	1-11
2–1	CSL Directory Structure	2-24
4–1	CACHE Primary Summary	4-3
4–2	CACHE CSL Macros Using CACHE Port Number	4-8
5–1	CHIP Functions	5-2
6–1	DAT Primary Summary	6-2
7–1	DMA Primary Summary	7-3
7–2	DMA CSL Macros Using DMA Port Number	7-11
7–3	DMA CSL Macros Using Handle	7-12
8–1	EMIF Primary Summary	8-3
8–2	EMIF CSL Macros using EMIF Port Number	8-10
9–1	GPIO CSL Macros Using GPIO Port Number	9-3
10–1	IRQ Configuration Structure	10-2
10–2	IRQ Functions	10-3
10–3	IRQ_EVT_NNNN Event List	10-4
11–1	McBSP Primary Summary	11-3
11–2	MCBSP Macros Using MCBSP Port Number	11-12
11–3	McBSP CSL Macros Using Handle	11-13
12–1	PLL Primary Summary	12-3
12–2	PLL CSL Macros Using PLL Port Number	12-8
13–1	PWR Functions	13-2
13–2	PWR CSL Macros Using PWR Port Number	13-4
14–1	TIMER Primary Summary	14-3
14–2	TIMER CSL Macros Using Timer Port Number	14-9
14–3	TIMER CSL Macros Using Handle	14-10

Examples

1–1 Using PER_Config . . . 1-13
1–2 Using PER_ConfigArgs . . . 1-13
2–1 Properties Page Options . . . 2-9
2–2 Modifying the C File . . . 2-14
2–3 Initializing a DMA Channel with DMA_config() . . . 2-20
2–4 Initializing a DMA Channel with DMA_configArgs() . . . 2-22
2–5 Using a Linker Command File . . . 2-30
3–1 CACHE Header File . . . 3-6
3–2 CACHE Source File (Declaration Section) . . . 3-7
3–3 CACHE Source File (Body Section) . . . 3-7
3–4 DMA Header File . . . 3-12
3–5 DMA Source File (Declaration Section) . . . 3-13
3–6 DMA Source File (Body Section) . . . 3-13
3–7 EMIF Header File . . . 3-18
3–8 EMIF Source File (Declaration Section) . . . 3-18
3–9 EMIF Source File (Body Section) . . . 3-19
3–10 GPIO Source File (Body Section) . . . 3-22
3–11 MCBSP Header File . . . 3-27
3–12 MCBSP Source File (Declaration Section) . . . 3-28
3–13 MCBSP Source File (Body Section) . . . 3-29
3–14 PLL Header File . . . 3-33
3–15 PLL Source File (Declaration Section) . . . 3-34
3–16 PLL Source File (Body Section) . . . 3-34
3–17 Timer Header File . . . 3-39
3–18 Timer Source File (Declaration Section) . . . 3-40
3–19 Timer Source File (Body Section) . . . 3-40
11–1 McBSP Port Initialization Using MCBSP_config() . . . 11-14

Chapter 1

CSL Overview

This chapter introduces the Chip Support Library, briefly describes its architecture, and provides a generic overview of the collection of functions, macros, and constants that help you program DSP peripherals.

Topic **Page**

1.1	Introduction to CSL	1-2
1.2	Naming Conventions	1-5
1.3	Data Types	1-6
1.4	Symbolic Constant Values	1-8
1.5	Macros	1-9
1.6	Functions	1-11

1.1 Introduction to CSL

The Chip Support Library(CSL) is a fully scalable component of DSP/BIOS that provides C-program functions to configure and control on-chip peripherals. It is intended to simplify the process of running algorithms in a real system. The goal is peripheral ease of use, shortened development time, portability, hardware abstraction, and a small level of standardization and compatibility among devices.

How the CSL Benefits You

- **Standard Protocol to Program Peripherals**

  A standard protocol to each programming of on-chip peripherals. This includes data types and macros to define a peripherals configuration, and functions to implement the various operations of each peripheral.

- **Basic Resource Management**

  Basic resource management is provided through the use of open and close functions for many of the peripherals. This is especially helpful for peripherals that support multiple channels.

- **Symbol Peripheral Descriptions**

  As a side benefit to the creation of CSL, a complete symbolic description of all peripheral registers and register fields has been created. It is suggested that you use the higher level protocols described in the first two benefits, as these are less device specific, thus making it easier to migrate your code to newer versions of DSPs.

CSL integrates GUI, graphic user interface, into the DSP/BIOS configuration tool. The CSL tree of the configuration tool allows the pre-initialization of some peripherals by generating C files using CSL APIs. The peripherals are pre-configured with the pre-defined configuration objects (see Chapter 2, *How To Use CSL)*.

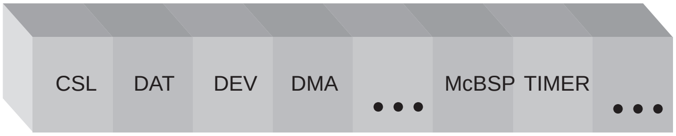
Chapter 3, *DSP/BIOS Configuration Tool: CSL Modules*, details the available CSL modules found in the DSP/BIOS Configuration tool.

CSL Architecture

The CSL consists of discrete modules that are built and archived into a library file. Each peripheral is covered by a single module while additional modules provide general programming support.

Figure 1–1 illustrates the individual API modules. This architecture allows for future expansion because new modules can be added as new peripherals emerge.

Figure 1–1. API Modules



Although each API module provides a unique API, some interdependency exists between the modules. For example, the DMA module depends on the IRQ module because of DMA interrupts; As a result, when you link code that uses the DMA module, a portion of the IRQ module is linked automatically.

Each module has a compile-time support symbol that denotes whether or not the module is supported for a given device. For example, the symbol _DMA_SUPPORT has a value of 1 if the current device supports it and a value of 0 otherwise. The available symbols are located in Table 1–1. You can use these support symbols in your application code to make decisions.

Table 1–1 lists general and peripheral modules with their associated include file and the module support symbol, that should be included in your application.

Table 1–1. CSL Modules and Include Files

Peripheral Module (PER)	Description	Include File	Module Support Symbol
CACHE	Cache	csl_cach.h	_CACHE_SUPPORT
CHIP	Device specific module	csl_chip.h	_CHIP_SUPPORT
DAT	Device independent data copy/fill module	csl_dat.h	_DAT_SUPPORT
DMA	Direct memory access	csl_dma.h	_DMA_SUPPORT
EMIF	External memory bus interface	csl_emif.h	_EMIF_SUPPORT
GPIO	General purpose I/O	csl_gpio.h	_GPIO_SUPPORT
IRQ	Interrupt controller	csl_irq.h	_IRQ_SUPPORT
MCBSP	Multi-channel buffered serial port	csl_mcbsp.h	_MCBSP_SUPPORT
PLL	PLL	csl_pll.h	_PLL_SUPPORT
PWR	Power-down	csl_pwr.h	_PWR_SUPPORT
TIMER	Timer peripheral	csl_timer.h	_TIMER_SUPPORT

Table 1–2 lists the C5000 devices that CSL supports and the Large and Small-Model libraries included in CSL. The device support symbol to be used with the compiler.

Table 1–2. CSL Device Support

Device	Small-Model Library	Large-Model Library	Device Support Symbol
C5510	csl5510.lib	csl5510x.lib	CHIP_5510
C5510 PG 1.0	csl5510 pg1.lib	csl5510pg1x.lib	CHIP_5510PG1_0
C5510 PG 2.0	csl5510 pg2.lib	csl5510pg2x.lib	CHIP_5510PG2_0

1.2 Naming Conventions

The following conventions are used when naming CSL functions, macros and data types:

Table 1–3. CSL Naming Conventions

Object Type	Naming Convention
`Function`	`PER_funcName()`†
`Variable`	`PER_varName()`†
`Macro`	`PER_MACRO_NAME`†
`Typedef`	`PER_Typename`†
`Function Argument`	`funcArg`
`Structure Member`	`memberName`

† PER is the placeholder for the module name.

- All functions, macros and data types start with PER_ (where PER is the Peripheral module name listed in Table 1–1) in capital letters.
- Function names use all small letters. Capital letters are used only if the function name consists of two separate words. (for example, PER_getConfig()).
- Macro names use all capital letters (for example, DMA_DMPREC_RMK).
- Data types start with a capital letter followed by small letters (for example, DMA_Handle).

1.3 Data Types

The CSL provides its own set of data types that all begin with a capital letter. Table 1–4 lists the CSL data types as defined in the stdinc.h file.

Table 1–4. CSL Data Types

Data Type	Description
Bool	unsigned short
*PER*_Handle	void *
Int16	short
Int32	long
Uchar	unsigned char
Uint16	unsigned short
Uint32	unsigned long
DMA_AdrPtr	void (*DMA_AdrPtr)() pointer to a void function

1.3.1 Resource Management

CSL provides a limited set of functions to enable resource management for applications that support multiple algorithms and may reuse the same peripheral device.

Resource management in CSL is achieved through API calls to the PER_open and PER_close functions. The PER_open function normally takes a device number and reset flag as the primary arguments and returns a pointer to a Handle structure that contains information about which channel (DMA) or port (MCBSP) was opened. When given a specific device number, the open function checks a global flag to determine its availability. If the device/channel is available, then it returns a pointer to a predefined Handle structure for this device. If the device has already been opened by another process, then an invalid Handle is returned with a value equal to the CSL symbolic constant, INV.

Note: To ensure that no resource usage conflicts occur, CSL performs other function calls, other than returning an invalid handle from the PER_open function. You must check the value returned from the PER_opne function to guarantee that the resource has been allocated.

Before accepting a handle object as an argument, API functions first check to ensure that a valid Handle has been passed.

Calling PER_close frees a device/channel for use by other processes. PER_close clears the in_use flag and resets the device/channel.

All CSL modules that support multiple devices or channels, such as MCBSP and DMA, require a device Handle as primary argument to most API functions. For these APIs , the definition of a PER_Handle object is required.

1.3.1.1 Using CSL Handles

CSL Handle objects are used to uniquely identify an opened peripheral channel/port or device. Handle objects must be declared in the C source, and initialized by a call to a PER_open function before calling any other API functions that require a handle object as argument.
For example:

```
DMA_Handle myDma;  /* Defines a DMA_Handle object, myDma */
```

Once defined, the CSL Handle object is initialized by a call to PER_open:

```
.
.
myDma = DMA_open(DMA_CHA0,DMA_OPEN_RESET); /* Open DMA channel
0 */
```

The call to DMA_open initializes the handle, myDma. This handle can then be used in calls to other API functions:

```
DMA_start(myDma);          /* Begin transfer */
.
.
.
DMA_close(myDma);           /* Free DMA channel */
```

Note: Handles are required only for peripherals that have multiple channels or ports, such as DMA, MCBSP, TIMER, and DAT.

1.4 Symbolic Constant Values

To facilitate initialization of values in your application code, the CSL provides symbolic constants for registers and writable field values as described in Table 1–5 The following naming conventions are used:

- *PER* indicates a peripheral module as listed in Table 1–1 on page 1-3.
- *REG* indicates a peripheral register.
- *FIELD* indicates a field in the register.
- *SYMVAL* indicates the symbolic value of a register field.

Table 1–5. Generic CSL Symbolic Constants

(a) Constant Values for Registers

Constant	Description
*PER_REG_***DEFAULT**	Default value for a register; corresponds to the register value after a reset or to 0 if a reset has no effect.

(b) Constant Values for Fields

PER_REG_FIELD_SYMVAL	Symbolic constant to specify values for individual fields in the specified peripheral register.
*PER_REG_FIELD_***DEFAULT**	Default value for a field; corresponds to the field value after a reset or to 0 if a reset has no effect.

1.5 Macros

Table 1–6 provides a generic description of the most common CSL macros. The following naming conventions are used:

- *PER* indicates a peripheral module as listed in Table 1–1 on page 1-3.
- *REG#* indicates, if applicable, a register with the channel number. (For example: DMAGCR, TCR0, ...)
- *FIELD* indicates a field in a register.
- *regval* indicates an integer constant, an integer variable, a symbolic constant (*PER_REG*_DEFAULT), or a merged field value created with the *PER*_REG_RMK() macro.
- *fieldval* indicates an integer constant, integer variable, macro, or symbolic constant (*PER_REG_FIELD_SYMVAL*) as explained in section 1.4; all field values are right justified.

CSL also offers equivalent macros to those listed in Table 1–6, but instead of using REG# to identify which channel the register belongs to, it uses the Handle value. The Handle value is returned by the PER_open() function. These macros are shown Table 1–7. Please note that REG is the register name *without the channel number*.

Table 1–6. Generic CSL Macros

Macro	Description
*PER*_REG_**RMK**(, *fieldval_15,* . . . *fieldval_0*)	Creates a value to store in the peripheral register; _RMK macros make it easier to construct register values based on field values. The following rules apply to the _RMK macros: ❑ Defined only for registers with more than one field. ❑ Include only fields that are writable. ❑ Specify field arguments as most-significant bit first. ❑ Whether or not they are used, all writable field values must be included. ❑ If you pass a field value exceeding the number of bits allowed for that particular field, the _RMK macro truncates that field value.
*PER*_**RGET**(*REG#*)	Returns the value in the peripheral register.
*PER*_**RSET**(*REG#,* *regval*)	Writes the value to the peripheral register.
*PER*_**FMK** *(REG,* FIELD*,* *fieldval)*	Creates a shifted version of *fieldval* that you could OR with the result of other _FMK macros to initialize register REG. This allows you to initialize few fields in REG as an alternative to the _RMK macro that requires that ALL the fields in the register be initialized.

Table 1–6. Generic CSL Macros (Continued)

Macro	Description
*PER*_**FGET**(*REG#, FIELD*)	Returns the value of the specified *FIELD* in the peripheral register.
*PER*_**FSET**(*REG#*, *FIELD, fieldval*)	Writes *fieldval* to the specified *FIELD* in the peripheral register.
*PER*_**ADDR**(*REG#*)	If applicable, gets the memory address (or sub-address) of the peripheral register REG#.

Table 1–7. Generic CSL Macros (Handle-based)

Macro	Description
*PER*_**RGET_H**(handle, *REG*)	Returns the value of the peripheral register REG associated with Handle.
*PER*_**RSET_H**(handle, *REG, regval*)	Writes the value to the peripheral register REG associated with Handle.
*PER*_**ADDR_H**(handle, *REG*)	If applicable, gets the memory address (or sub-address) of the peripheral register REG associated with Handle.
*PER*_**FGET_H**(handle, *REG, FIELD*)	Returns the value of the specified *FIELD* in the peripheral register REG associated with Handle.
*PER*_**FSET_H**(handle, *REG*, *FIELD,* fieldval)	Sets the value of the specified *FIELD* in the peripheral register REG to fieldval.

1.6 Functions

Table 1–8 provides a generic description of the most common CSL functions where *PER* indicates a peripheral module as listed in Table 1–1 on page 1-3. **Because not all of the functions are available for all the modules, specific descriptions and functions are listed in each module chapter.** The following conventions are used in Table 1–8:

- Italics indicate variable names.
- Brackets [...] indicate optional parameters.
  - *[handle]* is required only for the handle-based peripherals: DAT, DMA, MCBSP, and TIMER. See Section 1.3.1.1 on page 1-7, *Using CSL Handles*.
  - *[priority]* is required only for the DAT peripheral module.

Table 1–8. Generic CSL Functions

Function	Description
handle = *PER*_**open**(*channelNumber,* *[priority]* *flags*)	Opens a peripheral channel and then performs the operation indicated by *flags*; must be called before using a channel. The return value is a unique device handle to use in subsequent API calls. The *priority* parameter applies only to the DAT module.
*PER*_**config**(*[handle,]* **configStructure*)	Writes the values of the configuration structure to the peripheral registers. You can initialize the configuration structure with: ❑ Integer constants ❑ Integer variables ❑ CSL symbolic constants, *PER_REG*_DEFAULT (See Section 1.4 on page 1-8, CSL *Symbolic Constant Values*) ❑ Merged field values created with the *PER_REG*_RMK macro
*PER*_**configArgs**(*[handle,]* *regval_1,* . . . *regval_n*)	Writes the individual values (*regval_n*) to the peripheral registers. These values can be any of the following: ❑ Integer constants ❑ Integer variables ❑ CSL symbolic constants, *PER_REG*_DEFAULT ❑ Merged field values created with the *PER_REG*_RMK macro

Table 1–8. Generic CSL Functions (Continued)

Function	Description
*PER*_**start**(*[handle])* *[txrx],* *[delay]* *)*	Starts the peripheral after using PER_config() or PER_configArgs(). [txrx] and [delay] apply only to MCBSP.
*PER*_**reset**(*[handle]* *)*	Resets the peripheral to its power-on default values.
*PER*_**close**(*handle*)	Closes a peripheral channel previously opened with *PER*_open(). The registers for the channel are set to their power-on defaults, and any pending interrupt is cleared.

1.6.1 Initializing Registers

The CSL provides two types of functions for initializing the registers of a peripheral: *PER*_config and *PER*_configArgs (where *PER* is the peripheral as listed in Table 1–1 on page 1-3).

- *PER*_config allows you to initialize a configuration structure with the appropriate register values and pass the address of that structure to the function, which then writes the values to the register. Example 1–1 shows an example of this method.
- *PER*_configArgs allows you to pass the individual register values as arguments to the function, which then writes those individual values to the register. Example 1–2 shows an example of this method.

You can use these two initialization functions interchangeably, but you still need to generate the register values. To simplify the process of defining the values to write to the peripheral registers, the CSL provides the PER_REG_RMK (make) macros, which form merged values from a list of field arguments. Macros are covered in Section 1.5, on page 1-9, *CSL Macros*.

Example 1–1. Using PER_Config

```
PER_Config MyConfig = {
  reg0,
  reg1,
  …
};
main() {
…
PER_config(&MyConfig);
…
}
```

Example 1–2. Using PER_ConfigArgs

```
PER_configArgs(reg0, reg1, …);
```

Chapter 2

How to Use CSL

This chapter provides instructions and examples that explain the configuration and use of CSL DSP/BIOS. Specific examples are provided in each module chapter.

Topic **Page**

Topic		Page
2.1	Installing the Chip Support Library	2-2
2.2	Overview	2-3
2.3	DSP/BIOS Configuration Tool: CSL Tree	2-4
2.4	Generation of the C Files (CSL APIs)	2-8
2.5	Creating a Configuration	2-12
2.6	Example of CSL APIs Generation (TIMER Module)	2-15
2.7	Configuring Peripherals Without GUI	2-20
2.8	Compiling and Linking With CSL	2-24
2.9	Rebuilding CSL	2-31
2.10	Using Function Inlining	2-31

2.1 Installing the Chip Support Library

Code Composer Studio™ (CCS) release version 2.0 and greater automatically installs the CSL. If you are using an earlier version of CCS, follow these steps to install CSL:

1) Unzip csl.zip into a temporary folder.

2) Copy all C header files (*.h) into c:\ti\c5500\bios\include

3) Copy all library files (*.lib) into c:\ti\c5500\bios\lib

2.2 Overview

With a few exceptions (GPIO, PLL), all of the CSL module functions operate on two types of objects:

- The PER_Handle object
- The PER_Config object

These objects are predefined C structure types which when properly declared and initialized, contain all the information necessary to configure and control the peripheral device.

There are two ways to configure peripherals when using CSL. One is manual configuration by declaring and initializing objects and the C source.

The other option is by using the DSP/BIOS Configuration Tool. This method is preferred because the graphical user interface provision that is part of the DSP/BIOS configuration tool is integrated into Code Composer Studio.

The CSL GUI provides the benefit of a visual tool that allows you to view the chosen register settings, determine which flags/options have been set by a particular mode selection, and most importantly, it is possible to have the code for the configuration settings automatically be created and stored in a C source file that can be integrated directly into your application.

2.3 DSP/BIOS Configuration Tool: CSL Tree

The DSP/BIOS Configuration Tool allows you to access the CSL graphical interface and configure some of the on-chip peripherals. Each peripheral is represented as a subdirectory of the CSL Tree as shown in Figure 2–1.

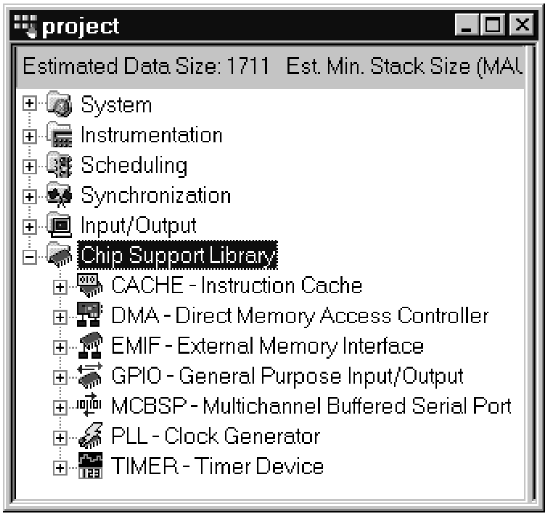
The work-flow consists of the following three main steps:

1) Creation of the DSP/BIOS configuration file (.cdb file). In Code Composer Studio, select File –> New –> DSP/BIOS Configuration.

2) Configuration of the on-chip peripherals by the user through the CSL hierarchy tree.

3) Automatic generation of the C-code files when saving the configuration file.

Figure 2–1. CSL Tree

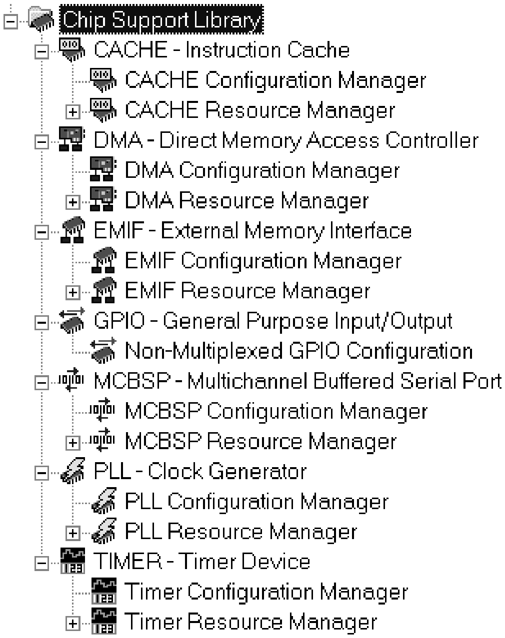


For the TMS320C5500 DSP platform, the peripherals available in the DSP/BIOS Configuration Tool are:

- CACHE
- DMA
- EMIF
- GPIO
- MCBSP
- PLL
- TIMER

Figure 2–2 shows an example of an expanded CSL Tree.

Figure 2–2. Expanded CSL Tree



Each peripheral is organized into several sections (see Figure 2–2):

- *PERIPHERAL* **Configuration Manager** – Allows you to set the peripheral register values by selecting the options through the Properties pages. Several configuration objects can be created by selecting the InsertdmaCfg option from the right-click menu (see Figure 2–3). The menu options allow you to rename and delete the configuration object (see Figure 2–4), and to display the Dependency Dialog box that allows you to determine which peripheral is using the configuration (see Figure 2–5).

- *PERIPHERAL* **Resource Manager** – Allows you to allocate the on-chip device which will be used like a DMA channel, a MCBSP port , or a TIMER device. The handle objects can be renamed only (no deletions permitted).

  The devices are displayed as pre-defined objects and cannot be deleted or renamed. However, the Handles to these objects can be renamed.

Figure 2–3. Insert Configuration Object

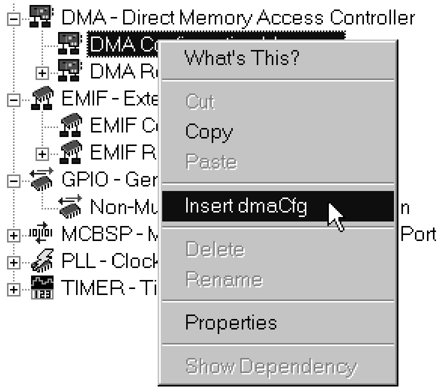


Figure 2–4. Delete/Rename Options

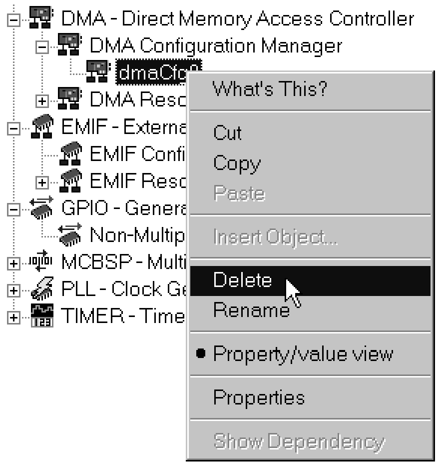
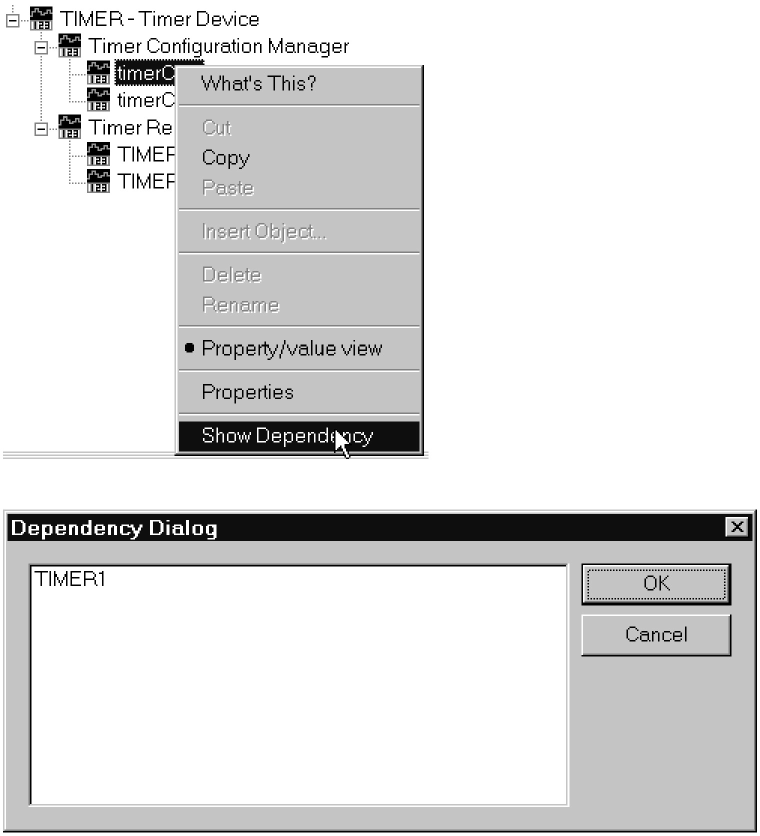


Figure 2–5. Show Dependency Option



2.4 Generation of the C Files (CSL APIs)

After saving the configuration file *project***.cdb**, the following C files are generated:

- Header file: *project***cfg.h**
- Source file: *project***cfg_c.c**

In these examples, *project* is your .cdb file name. The bold characters are attached automatically.

2.4.1 Header File *project*cfg.h

The header file contains several elements:

- The definition of the chip. For example, if the selected chip is 5510, the definition is:

  ```
  #define CHIP_5510 1
  ```

- The csl header files of the CSL tree

  ```
  #include <csl_dma.h>
  #include <csl_emif.h>
  #include <csl_mcbsp.h>
  #include <csl_timer.h>
  ```

- The declaration list of the *variables* Handle and configuration names defined in the *project*.cdb. These are declared external, as shown below:

  ```
  extern TIMER_Config timerCfg1;
  extern MCBSP_Config MCBSPmcbspCfg0;

  extern TIMER_Handle hTimer1;
  extern MCBSP_Handle hMcbsp0;
  ```

In order to access the predefined Handle and configuration objects, the header file must also be included in your project C file.

```
/* User's main .c file */
```

The following line is mandatory and must be included in the user's C file:

```
#include <projectcfg.h>
```

2.4.2 Source File *project*cfg_c.c

The source file consists of the Include section, the Declaration section, and the Body section.

- **Include** section:

  This section defines the header file. The source file has access to the data declared in the header file.

  `#include <projectcfg.h>`

Note: Note: If this line is added before the other csl header files (csl_emif, csl_timer, ...), you are not required to specify the device number under the Project option (that –dCHIP_55xx is not required).

- **Declaration** section:

  This section defines the configuration structures and the Handle objects previously defined in the configuration tool.

  The values of the registers reflect the options selected through the Properties pages of each device, as shown in Example 2–1.

Example 2–1. Properties Page Options

```
/*  Config Structures */
TIMER_Config timerCfg0 = {
    0x0010,         /*  Timer Control Register (TCR)   */
    0x0000,         /*  Timer Period Register (PRD)   */
    0x0000          /*  Timer Prescaler Register (PRSC)   */
};


DMA_Config dmaCfg0 = {
    0x0000,         /*  Source Destination Register (CSDP)  */
    0x0000,         /*  Control Register (CCR)  */
    0x0000,         /*  Interrupt Control Register (CICR)  */
    NULL,           /*  Lower Source Address (CSSA_L) – Symbolic(Byte Address)
*/
    NULL,           /*  Upper Source Address (CSSA_U) – Symbolic(Byte Address)
*/
    NULL,           /*  Lower Destination Address (CDSA_L) – Symbolic(Byte Ad-
dress)  */
    NULL,           /*  Upper Destination Address (CDSA_U) – Symbolic(Byte Ad-
dress)  */
    0x0001,         /*  Element Number (CEN)  */
    0x0001,         /*  Frame Number (CFN)  */
    0x0000,         /*  Frame Index (CFI)  */
    0x0000          /*  Element Index (CEI)  */
};



/*  Handles  */
TIMER_Handle hTimer1;
DMA_Handle hDma0;
```

- **Body** section

  The body is composed of a unique function, CSL_cfgInit(), which is called from your C file.

  The function CSL_cfgInit() allows you to allocate/open and configure a device by calling the Peripheral_open() and Peripheral_config() APIs, respectively.

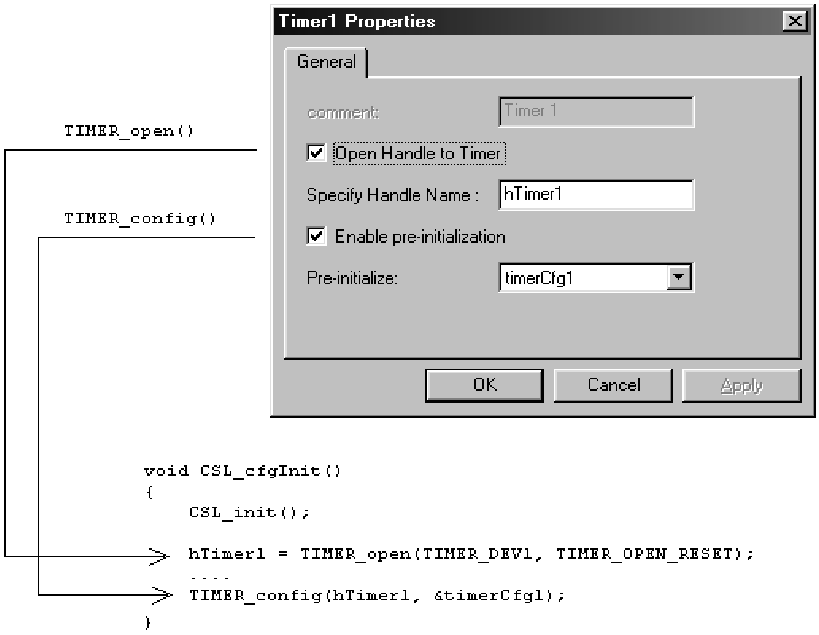
  These two functions are generated when the Open Handle to Timer and Enable pre-initialization options are checked in the Properties page of the related Resource Manager (see Figure 2–6).

Note: Note: A device can be allocated/opened without being configured.

In the example shown in Figure 2–6,

- If Enable pre-initialization is checked, the TIMER_config() function is generated.
- If Enable Pre-initialization is unchecked, TIMER_config() is not generated, but the configuration structure timerCfg1 is created and available for you to use.

Figure 2–6. Resource Manager Properties Page



Before using these predefined APIs, CSL_cfgInit must be called. This function is automatically called by the DSP/BIOS CSL boot/start-up routine.

```
/* User's file main.c */
void main ()
{
```

2.5 Modifying the Project Folder

To create a configuration, you must:

1) Modify the Project folder on the Code Composer Studio Interface

2) Modify the C code (main.c).

3) In Code Composer Studio, select File → New → DSP/BIOS Configuration: open Config1.cdb window (default name)

4) Select File → Save as: *project*.cdb (user cdb name)

5) Select Project → Add Files to Project: *project*.cdb (the files *project*cfg.s55 and *projectcfg_c.c* will appear in “generated files” folder)

6) Configure the CSL peripherals Properties pages as needed: Create the configuration objects and Opening of Handles objects. (see section 2.3 and section 2.4.2).

7) Save *project*.cdb

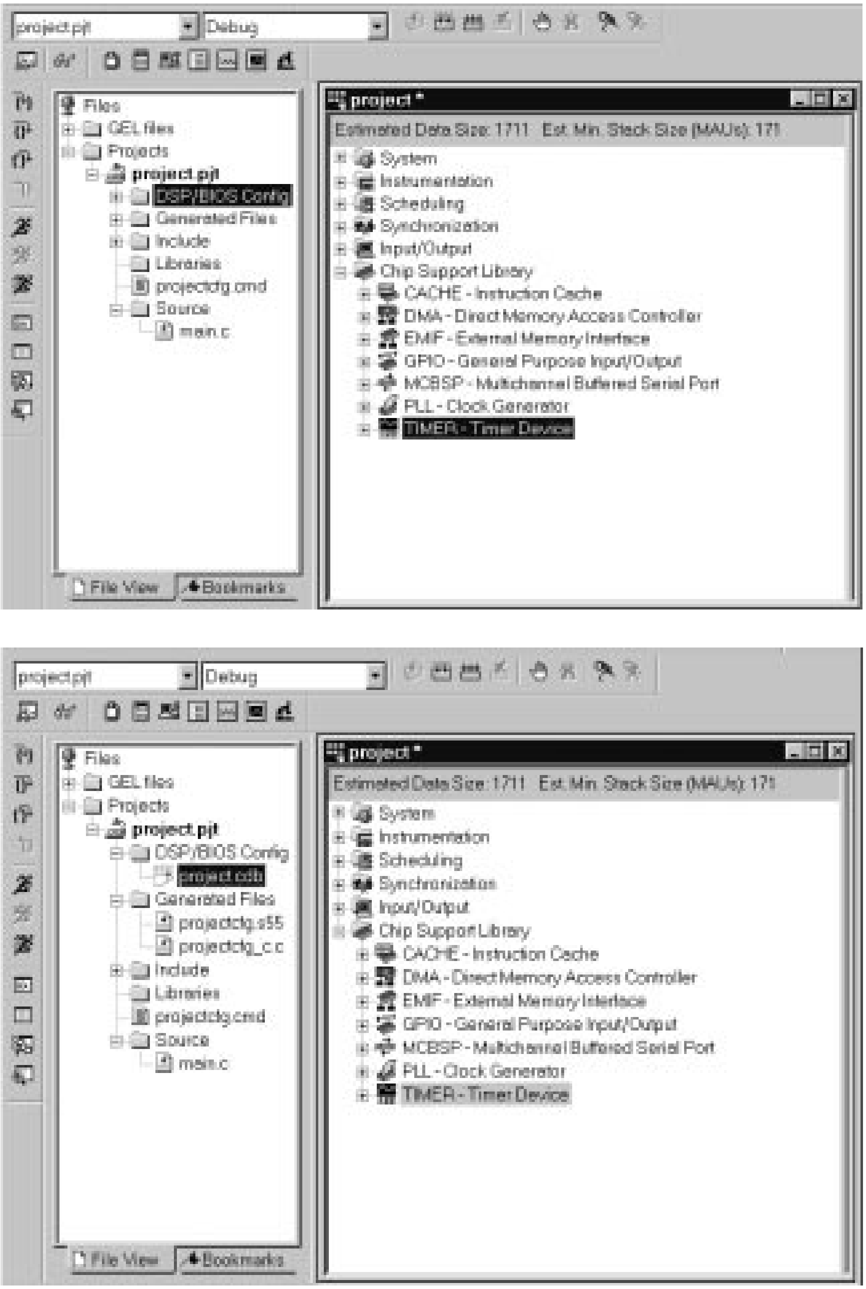
8) Select Project → Add Files to Project

9) Include the following files in your Project:

   - command file: *project*cfg.cmd
   - asm source file: *project.s55* (CSL predefined APIs)

Figure 2–7 shows the project layout after a .cdb file is created and the *project*.cmd, *project*.s55, and *projectcfg_c*.c files have been added to the project.

Figure 2–7. Practice Summary



2.5.1 Modification of C code (main.c)

To modify the C code (main.c):

1) Add the header file ***#include projectcfg.h*** to your main.c file, As shown In Example 2–2. These lines are required to provide access to the Handle and configuration objects.

Note: CSL_cfgInit() is automatically called by the DSP/BIOS CSL boot/start-up routine. This function pre-opens and pre-configures the peripherals **ONLY**. It does not start device operation. A call to the PER_Start function is required within your code to begin peripheral operation with the pre-chosen settings.

Example 2–2. Modifying the C File

```
/* Include file */
#include projectcfg.h

/* main program */
void main()
{

...
}
```

2.6 Example of CSL APIs Generation (TIMER Module)

This section provides an example using the 5510 device, which demonstrates how to open and define a configuration for a TIMER device using the graphical user interface. It also provides a full example of C files generated from a .cdb file by using the Chip Support Library APIs.

Warning:

First, go to Global Settings (System Folder) and select the chip type present on your board.

This step is very important because the chip type affects the setting of the default values of the peripheral registers. Make sure that you have not already created any configuration objects with the wrong chip type selected. Before switching chip types, it is recommended that you delete any existing configuration objects, which have default values that are not identical from one chip to another.

2.6.1 Configuration of the TIMER1 Device

The configuration file *mytimer*.cdb is assumed to be created previously and opened (see section 2.5, *Getting Started,* for more details).

In the CCS Project View window (see Figure 2–8) open *mytimer*.cdb, and go to the sub-folder TIMER module (CSL Folder).

Follow these steps:

1) Right-click on the TIMER Configuration Manager, insert a new configuration object.

2) Right-click on timerCfg0 and select Properties to open the timerCfg0 Properties window (as shown in Figure 2–9). Set the configuration by clicking on any of the tabs.

3) Under the Timer Resource Manager, right-click on Timer1 and select Properties to open the Timer1 Properties window (see Figure 2–9).

   - Check the Open Handle to Timer and Enable pre-initialization
   - From the pre-initialize drop-down list, select the configuration, timerCfg0.

Figure 2–8. CCS Project View

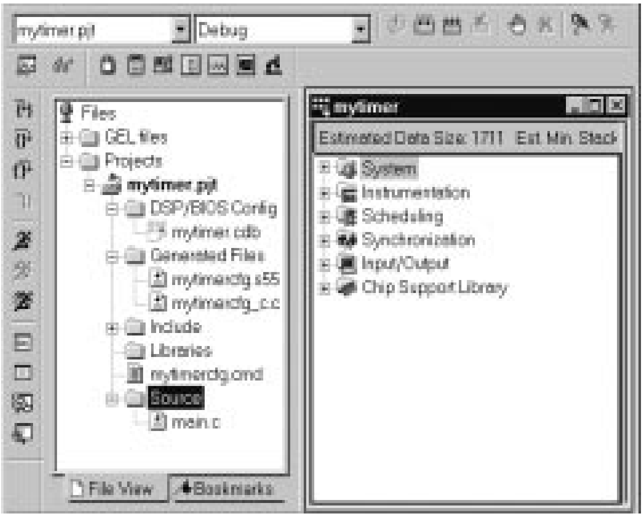
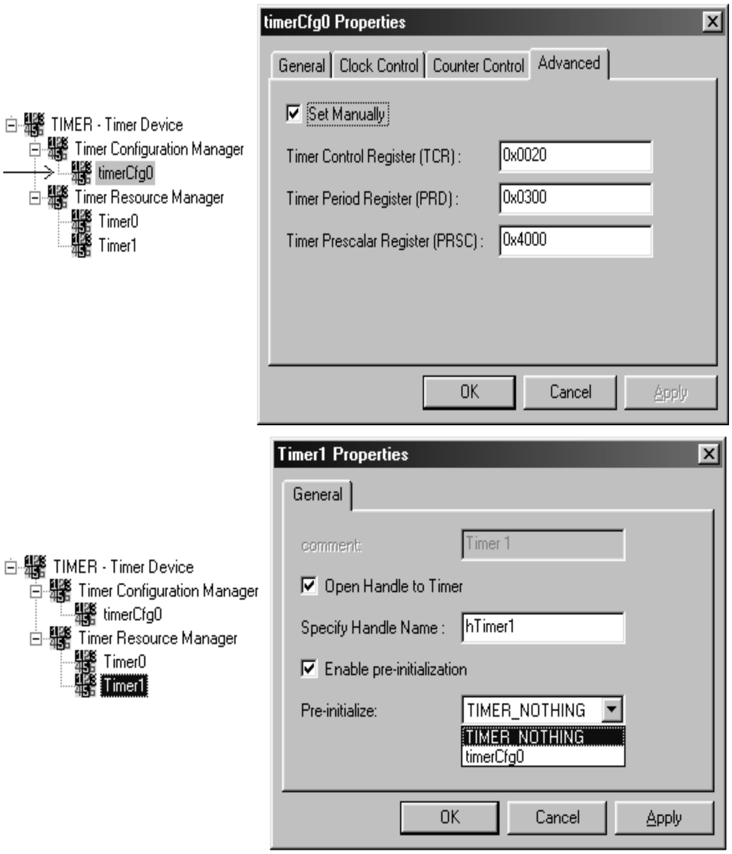


Figure 2–9. Configuring the TIMER1 Device



2.6.2 Generation of C Files

After saving the configuration file *mytimer*.cdb, the header file *mytimer*cfg.h and the source file *mytimer*cfg_c.c are generated (see Figure 2–10 and Figure 2–11).

Figure 2–10. Header File mytimercfg.h

```
/*   Do *not* directly modify this file.  It was    */
/*   generated by the Configuration Tool; any  */
/*   changes risk being overwritten.                */
/*  INPUT mytimer.cdb */
/*  Include Header Files  */
#include <std.h>
#include <hst.h>
#include <swi.h>
#include <tsk.h>
#include <log.h>
#include <sts.h>
#include <csl_timer.h>
#ifdef __cplusplus
extern "C" {
#endif
extern HST_Obj RTA_fromHost;
extern HST_Obj RTA_toHost;
extern SWI_Obj KNL_swi;
extern TSK_Obj TSK_idle;
extern LOG_Obj LOG_system;
extern STS_Obj IDL_busyObj;
extern TIMER_Config timerCfg0;
extern TIMER_Handle hTimer1;
extern void CSL_cfgInit();
#ifdef __cplusplus
}
#endif /* extern "C" */
```

csl header files of the peripherals implemented under the CSL tree

The Handle and Configuration objects are defined and can be used by other C files (User's files).

Figure 2–11. Source File mytimercfg_c.c

```
/*   Do *not* directly modify this file.  It was    */
/*   generated by the Configuration Tool; any  */
/*   changes risk being overwritten.                */

/*INPUT mytimer.cdb */

/*  Include Header File  */
#include <mytimercfg.h>

/*  Config Structures */
TIMER_Config timerCfg0 = {
    0x0020,         /*  Timer Control Register (TCR)   */
    0x0300,         /*  Timer Period Register (PRD)   */
    0x4000          /*  Timer Prescaler Register (PRSC)
*/
};

/*  Handles  */
TIMER_Handle hTimer1;

/*
 *  ======== CSL_cfgInit() ========
 */
void CSL_cfgInit()
{
    CSL_init();
   hTimer1 = TIMER_open(TIMER_DEV1, TIMER_OPEN_RESET);
   TIMER_config(hTimer1, &timerCfg0);
}
```

TIMER Configuration structure timerCfg0 with full TIMER peripheral register values

Handle **hTimer1** declaration

The **TIMER_open()** function returns the handle value in the handle variable hTimer1 previously declared.

TIMER_config() function sets the register values defined by the configuration object timerCfg0.

Figure 2–12. Example of main.c File Using Data Generated by the Configuration Tool

```
#include <csl.h>
#include <csl_timer.h>
#include <csl_irq.h>
#include <mytimercfg.h>

static Uint32 TIMEREventId1;

void main() {


  /* Obtain the event IDs for the TIMER devices */
  TIMEREventId1 = TIMER_getEventId(hTimer1);

  /* Enable the TIMER events */
  IRQ_enable(TIMEREventId1);

  /* Start the TIMERs – */
   TIMER_start(hTimer1);

  /* Waiting for TIMER Interrupt: */
    while( !IRQ_test(TIMEREventId1));

  /* Close TIMER */
  TIMER_close(hTimer1);

  }
```

This line is required and must be included in order to use the peripheral pre-initialization defined through the Configuration Tool.

Handle object “hTimer1” is used directly by the TIMER CSL APIs.

2.7 Configuring Peripherals Without GUI

Note: If you choose not to configure peripherals using GUI, you must pre-define the PER_Handle and PER_Config objects.

Example 2–3 illustrates the use of CSL to initialize DMA channel 0 and to copy a table from address 0x3000 to address 0x2000 using the _config() function. Example 2–4 is similar except that it uses the _configArgs() function.

Source address:	2000h in data space
Destination address:	3000h in data space
Transfer size:	Sixteen 16-bit single words

2.7.1 Using DMA_config()

Example 2–3 uses the DMA_config() function to initialize the registers.

Example 2–3. Initializing a DMA Channel with DMA_config()

```
// Step 1:   Include the
//           the header file of the module/peripheral you
//           will use <csl_dma.h>. The different header files are shown
//           in  Table 1–1.
//

#include <csl_dma.h>


// Example-specific initialization
#define N 16        // block size to transfer

#pragma DATA_SECTION(src,”table1”)   // scr data   table  address
Uint16 src[N] = {
       0xBEEFu,  0xBEEFu,  0xBEEFu,  0xBEEFu,
       0xBEEFu,  0xBEEFu,  0xBEEFu,  0xBEEFu,
       0xBEEFu,  0xBEEFu,  0xBEEFu,  0xBEEFu,
       0xBEEFu,  0xBEEFu,  0xBEEFu,  0xBEEFu
};

#pragma DATA_SECTION(dst, ”table2”)  // dst    data   table  address
Uint16 dst[N];
```

Example 2–3. Initializing a DMA Channel with DMA_config() (Continued)

```
//Step 2:    Define and initialize the DMA  channel
//           configuration structure

DMA_Config myconfig = {
   DMA_DMACSDP_RMK(0 , 0 ,0 ,0 , 0, 0 ,1),          /* DMACSDP  */
   DMA_DMACCR_RMK(1, 1, 0, 0, 0, 0,  0, 0, 0, 0),   /* DMACCR   */
   DMA_DMACICR(1, 1,1 , 1, 1, 1),                   /* DMACICR  */
   (DMA_AdrPtr) &src,                               /* DMACSSAL */
    0,                                              /* DMACSSAU */
    (DMA_AdrPtr)&dst,                               /* DMACDSAL */
    0,                                              /* DMACDSAU */
    N,                                              /* DMACEN   */
    1,                                              /* DMACFN   */
    0,                                              /* DMACFI   */
    0};                                             /* DMACEI   */

//Step 3:    Define a DMA_Handle pointer. DMA_open will initialize this handle
//           when a DMA channel is opened.

DMA_Handle myhDma;

void main(void) {

// .....

//Step 4:    Initialize the CSL Library. A one-time only initialization of the
//           CSL library must be done before calling any CSL module API.

   CSL_init();                      /* Init CSL     */

//Step 5: Open, configure and start the DMA channel.
//        To configure the channel you can use the
//        DMA_config() or DMA_configArgs() functions.

   myhDma = DMA_open(DMA_CHA0,0);   /* Open Channel           */
   DMA_config(myhDma, &myConfig);  /* Configure Channel      */
   DMA_start(myhDma);              /*  Begin Transfer         */

//Step 6: (Optional)
//        Use CSL DMA APIs to track DMA channel status.

   while(DMA_getStatus(myhDma));   /* Wait for complete */

//Step 7: Close DMA channel.

   DMA_close(myhDma);         /* Close channel (Optional) */
}
```

2.7.2 Using DMA_configArgs()

Example 2–4 performs the same task as Example 2–3 but uses DMA_configArgs() to initialize the registers.

Example 2–4. Initializing a DMA Channel with DMA_configArgs()

```
// Step 1:   Include the
//           the header file of the module/peripheral you
//           will use <csl_dma.h>. The different header files are shown
//           in  Table 1–1 on page 1-3.
//

#include <csl_dma.h>

// Example-specific initialization
#define N 16        // block size to transfer

#pragma DATA_SECTION(src,”table1”)   // scr data   table  address
Uint16 src[N] = {
       0xBEEFu,  0xBEEFu,  0xBEEFu,  0xBEEFu,
       0xBEEFu,  0xBEEFu,  0xBEEFu,  0xBEEFu,
       0xBEEFu,  0xBEEFu,  0xBEEFu,  0xBEEFu,
       0xBEEFu,  0xBEEFu,  0xBEEFu,  0xBEEFu
};

#pragma DATA_SECTION(dst, ”table2”)  // dst    data   table  address
Uint16 dst[N];

//Step 2:    Define a DMA_Handle pointer. DMA_open will initialize this handle
//           when a DMA channel is opened.

DMA_Handle myhDma;

void main(void) {

// .....

//Step 3:    Initialize the CSL Library. A One-time only initialization of the
//           CSL library must be done before calling any CSL module API.

   CSL_init();                        /* Init CSL      */
```

Example 2–4. Initializing a DMA Channel with DMA_configArgs() (Continued)

```
//Step 4: Open, configure and start the DMA channel.
//        To configure the channel you can use the
//        DMA_config() or DMA_configArgs() functions.

DMA_Config myconfig = {
   DMA_DMACSDP_RMK(0 , 0 ,0 ,0 , 0, 0 ,1),           /* DMACSDP  */
   DMA_DMACCR_RMK(1, 1, 0, 0, 0, 0,  0, 0, 0, 0),    /* DMACCR   */
   DMA_DMACICR(1, 1,1 , 1, 1, 1),                    /* DMACICR  */
   (DMA_AdrPtr) &src,                                /* DMACSSAL */
    0,                                               /* DMACSSAU */
    (DMA_AdrPtr)&dst,                                /* DMACDSAL */
    0,                                               /* DMACDSAU */
    N,                                               /* DMACEN   */
    1,                                               /* DMACFN   */
    0,                                               /* DMACFI   */
    0};                                              /* DMACEI   */

//Step 5: (Optional)
//        Use CSL DMA APIs to track DMA channel status.

   while(DMA_getStatus(myhDma));   /* Wait for complete */

//Step 6: Close DMA channel.

   DMA_close(myhDma);        /* Close channel */
}
```

2.8 Compiling and Linking With CSL

After writing your program, you have two methods available for compiling and linking your project:

- ❑ Use the DOS command line.
- ❑ Use the Code Composer Studio project build environment.

Table 2–1 lists the location of the CSL components after installation. Use this information when you set up the compiler and linker search paths. Section 2.8.3, *Creating a Linker File*, on page 2-30 explains specific requirements for the linker command file.

Table 2–1. CSL Directory Structure

This CSL component...	**Is located in this directory...**
Libraries	c:\ti\c5500\bios\lib
Source Library	c:\ti\c5500\bios\src
Include files	c:\ti\c5500\bios\include
Examples	c:\ti\examples\csl
Documentation	c:\ti\docs

2.8.1 Using the DOS Command Line

To compile and link your project using the DOS Command line:

1) Set the include file and library search paths.

Before you compile and link your program, you must verify that the include file search paths are correctly set for the compiler and that the library search path is correctly set for the linker. You can set these paths either in the autoexec.bat file or with the -i option.

- ❑ To set the include and library search paths, using the autoexec.bat file, add the following line to the autoexec.bat file:

```
SET C55x_C_DIR=.;C:\ti\c5500\bios\include;C:\ti\c5500\bios\lib;%C55x_C_DIR%
```

- ❑ To use the -i option, add the following when compiling and linking:

```
-i c:\ti\c5500\bios\include   (for the compiler)
-i c:\ti\c5500\bios\lib       (for the linker)
```

2) Select the correct C55x device and library to link to.

- To compile and link for near mode, type the following on the command line:

```
cl500 -dCHIP_5510 ex1.c  csl5510.lib linker.cmd -oex1.out
```

- To compile and link for far mode, type the following on the command line:

```
cl500 -mf -v558 -dCHIP_5510 ex1.c csl5510x.lib linker.cmd -oex1far.out
```

Notice the usage of the device support symbol CHIP_5510 (see Table 1–2 on page 1-4) to control conditional compilation. This usage is required because the C55x family offers different peripheral features that are specific to a particular C55x device.

2.8.2 Using the Code Composer Studio Project Environment

You must configure the CCS project environment to work with CSL. To configure the CCS Project environment, follow these steps listed below.

- Specify the target device you are using:

1) In Code Composer Studio, select Project→Options

2) In the Build Options dialog box, select the Compiler tab(see Figure 2–13)

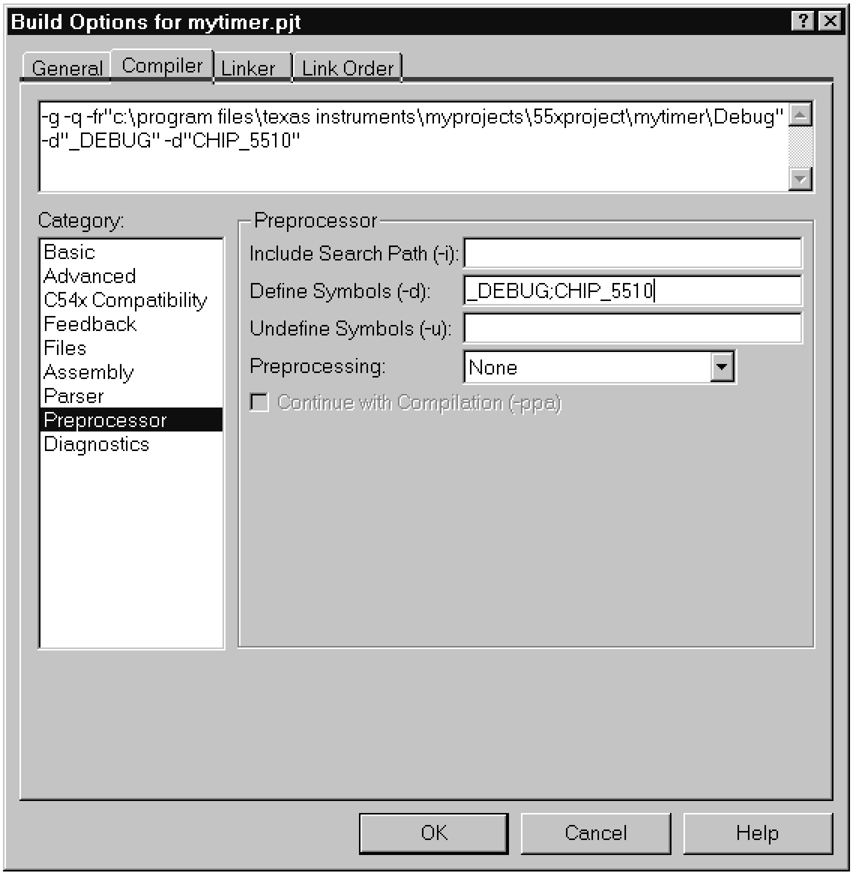
3) In the Category list box, highlight Preprocessor.

4) In the Define Symbols field, enter one of the device support symbols in Table 1–2, on page 1-4.

   For example, if you are using the 5510 device, enter CHIP_5510.

5) Click OK.

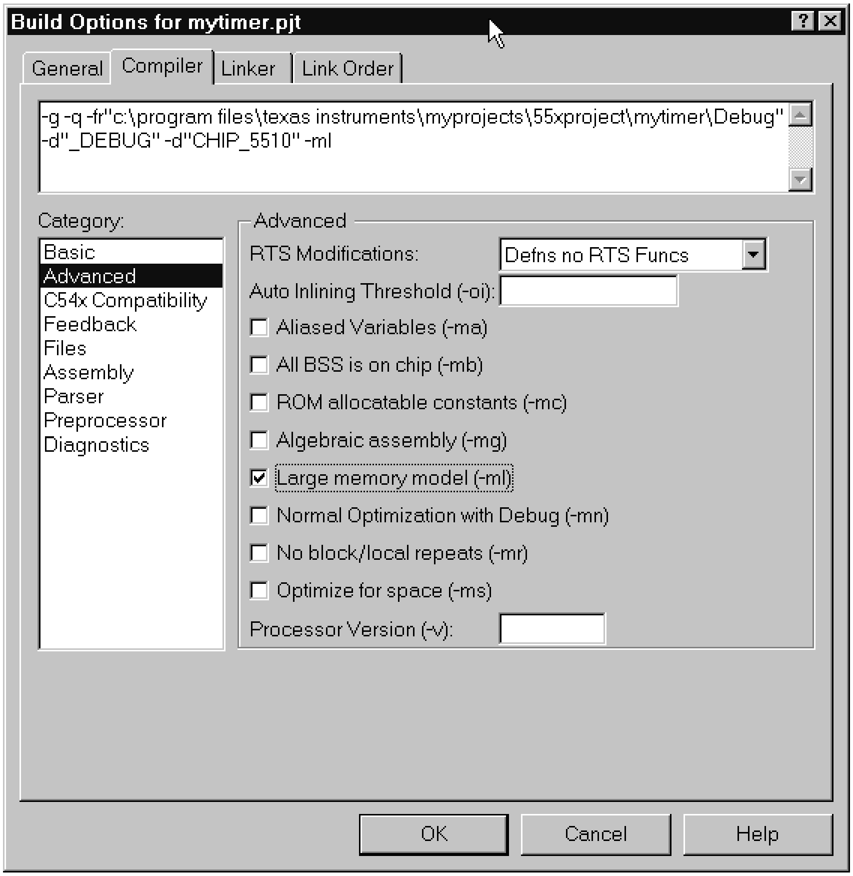
Figure 2–13. Defining the Target Device in the Build Options Dialog



❑ If you use any far-mode libraries, define far mode for the compiler and link with the far mode runtime library (rts_55x.lib):

1) In Code Composer Studio, select Project→Options
2) In the Build Options dialog box, select the Compiler Tab (Figure 2–14),
3) In the Category list box, highlight advanced.
4) Select Use Far Calls.
5) In the Processor Version (-v) field, type 548.
6) Click OK.

Figure 2–14. Defining Large Memory Model



- If you are using Code Composer Studio releases prior to 2.0, add the search path for the header files:

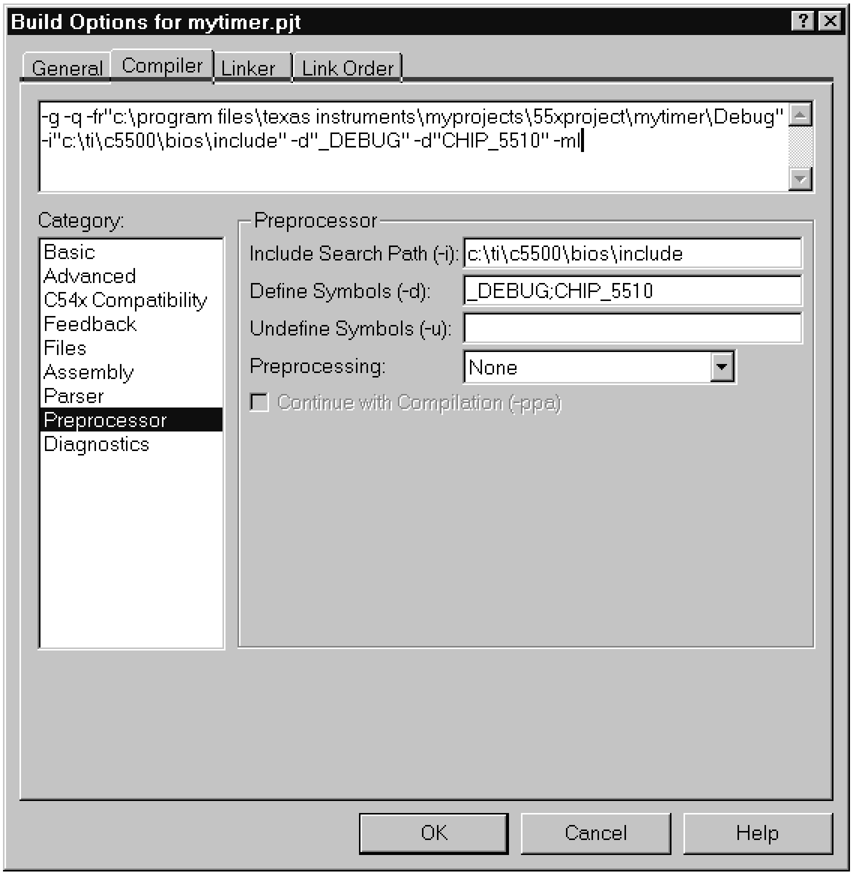
1) In Code Composer Studio, select Project→Options...

2) In the Build Options Dialog box, select the Compiler Tab (see Figure 2–15).

3) In the Include Search Path field (-i), type:
c:\ti\c5500\bios\include

4) Click OK.

Figure 2–15. Adding the Include Search Path

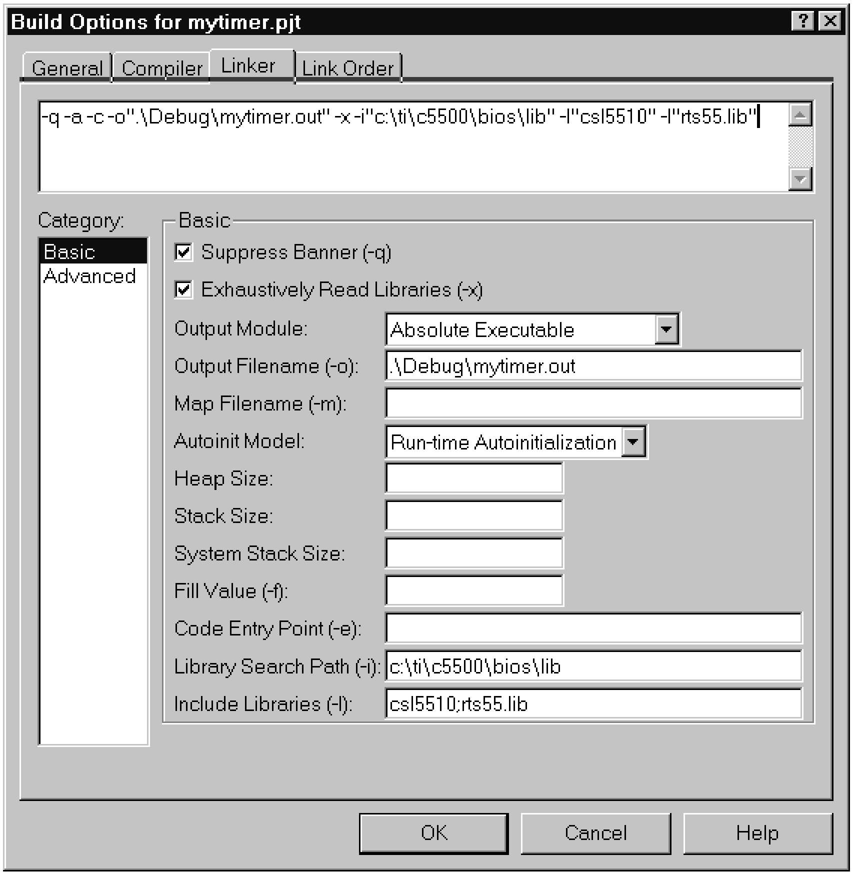


❑ Specify the search path for the CSL library:

1) In Code Composer Studio, select Project→Options
2) In the Build Options dialog box, Select the Linker Tab (see Figure 2–16).
3) In the Category list, highlight Basic.
4) In the Library search Path field (-l), type:
   c:\ti\c5500\bios\lib
5) In the Include Libraries (-i) field, enter the correct library from Table 1–2, on page 1-4.

   For example, if you are using the 5510 device, enter csl5510.lib for near mode or csl5510x.lib for far mode.
6) Click OK.

Figure 2–16. Defining Library Paths



2.8.3 Creating a Linker Command File

The CSL has two requirements for the linker command file:

- **You must allocate the .csl data section.**

  CSL creates a .csl data section to maintain global data that CSL uses to implement functions with configurable data. You must allocate this section within the base 64K address space of the data space.

- **You must reserve address 0x7b in scratch pad memory**

  The CSL uses address 0x7b in the data space as a pointer to the .csl data section, which is initialized during the execution of CSL_init(). For this reason, you must call CSL_init() before calling any other CSL functions. Overwriting memory location 0x7b can cause the CSL functions to fail.

Example 2–5 illustrates these requirements which must be included in the linker command file.

Example 2–5. Using a Linker Command File

```
MEMORY
{
      PROG0:    origin =  8000h, length = 0D000h
      PROG1:    origin = 18000h, length = 08000h

      DATA:    origin =  1000h, length = 04000h
}

SECTIONS
{
   .text   > PROG0
   .cinit  > PROG0
   .switch > PROG0
        .data   > DATA
   .bss    > DATA
   .const  > DATA
   .sysmem > DATA
   .stack  > DATA
   .csldata > DATA

   table1 : load =  6000h
   table2 : load =  4000h
}
```

2.9 Rebuilding CSL

All CSL source code is archived in the file csl55.src located in the c:\ti\bios\src\ folder. For example, to rebuild csl5510x.lib, type the following on the command line:

```
mk55  csl55.src -dCHIP5510 -ml
```

2.10 Using Function Inlining

Because some CSL functions are short, they set only a single bit field. In this case, incurring the overhead of a C function call is not always necessary. If you enable inline, the API declares these functions as *static inline*. Using this technique can help reduce code size. In order to allow for future changes, the CSL documentation does not identify which functions are inlined; however, if you enable function inlining with the compiler -x option, you see an increase in CSL code performance.

Chapter 3

DSP/BIOS Configuration Tool: CSL Modules

Note: In most cases, you are not required to use the DSP/BIOS™ configuration tool to configure peripherals.

The Chip Support Library (CSL) graphical user interface is part of the DSP/BIOS™ configuration tool integrated in Code Composer Studio (CCS). This graphical user interface (GUI) benefits you by reducing manual C-code generation and offering an easy way to use on-chip peripherals by programming the associated Peripheral registers through the properties pages.

Topic **Page**

3.1 Overview 3-2
3.2 CACHE Module 3-3
3.3 DMA Module 3-8
3.4 EMIF Module 3-14
3.5 GPIO Module 3-20
3.6 MCBSP Module 3-23
3.7 PLL Module 3-30
3.8 TIMER Module 3-35

3.1 Overview

Chapter 2 outlined the basic CSL program flow and illustrated the use of CSL macros in C source for declaring and defining the necessary PER_Handle and PER_Config objects needed for peripheral operation in CSL.

As an alternative to the manual declaration and initialization of the peripheral configuration objects within the C source described in chapter 2, CSL also provides a graphical user interface (GUI) that is part of the DSP/BIOS configuration tool and is integrated into Code Composer Studio.

The CSL graphical user interface (GUI) provides the benefit of a visual tool that allows you to view the chosen register settings, determine which flags/options have been set by a particular mode selection, and most importantly, have the code for the configuration settings automatically be created and stored in a C source file that can be integrated directly into your application.

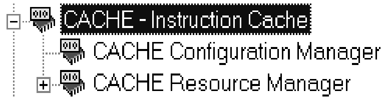
3.2 CACHE Module

3.2.1 Overview

The Cache module facilitates control of the program C55x program cache. The Cache module consists of a configuration manager and a resource manager. The configuration manager allows creation of one or more objects that contain data to completely configure all the Cache registers. The resource manager associates a specified configuration object with the Cache.

Figure 3–1 illustrates the CACHE sections menu on the CSL graphical user interface (GUI).

Figure 3–1. CACHE Sections Menu



The CACHE includes the following two sections:

- **CACHE Configuration Manager**: Allows you to create configuration objects. There are no predefined configuration objects.
- **CACHE Resource Manager**: Allows you to associate a configuration object to the cache.

3.2.2 CACHE Configuration Manager

The CACHE Configuration Manager allows you to create device configurations through the Properties page and to generate the configuration objects.

3.2.2.1 Creating/Inserting a Configuration Object

There is no predefined configuration object available. To configure the instruction cache through the peripheral registers, you must insert a new configuration object.

To insert a new configuration object, right-click on the CACHE Configuration Manager and select Insert cacheCfg from the drop-down menu. The configuration objects can be renamed. Their use depends on the on-chip device resources. Because only one instruction CACHE is available, no more than one configuration may be used at any time.

3.2.2.2 Deleting/Renaming an Object

To delete or rename an object, right-click on the configuration object to be deleted or renamed.

If a configuration object is used by one of the predefined handle objects of the CACHE Resource Manager (see Section 3.2.3, *CACHE Resource Manager*), the Delete and Rename options are grayed out and non-usable. The Show Dependency option is accessible and shows which device is using the configuration object. See Section 2.2, *Introduction to DSP/BIOS Configuration Tool: CSL Tree*, on page 2-3.

3.2.2.3 Configuring the Object Properties

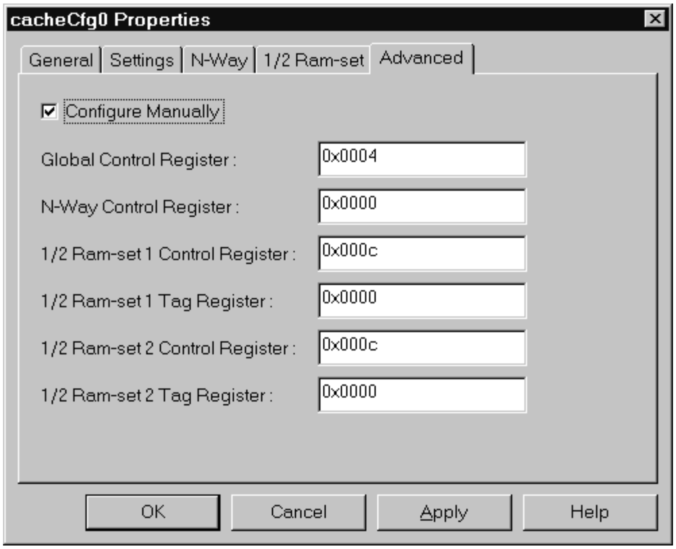
The Properties pages allow you to set the peripheral registers related to the CACHE. Each Tab page is composed of several options that are set to a default value (at device reset). The options represent the fields of the CACHE registers.

You can get the the configuration options through various properties pages as follows:

- ❑ Settings:Allows you to configure the Global Enable, Flush, and Ram Fill Mode
- ❑ N-Way:Allows you to configure N-Way settings
- ❑ 1/2 Ram-set:Allows you to configure 1/2 Ram-set settings
- ❑ Advanced: Summary of the previous pages. This page contains the full hexadecimal register values and reflects the option settings in the previous pages

Full register values can be entered directly and the new options are mirrored on the corresponding pages automatically.

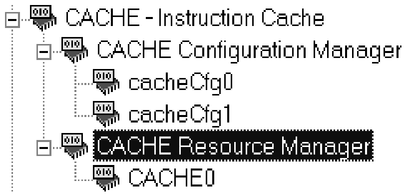
Figure 3–2. CACHE Properties Page



3.2.3 CACHE Resource Manager

The CACHE Resource Manager allows you to generate the CACHE_open() and the CACHE_config() CSL functions. Because there is only one CACHE supported, only one resource is available.

Figure 3–3. CACHE Resource Manager Menu



3.2.3.1 Properties Page

You can generate the CACHE_open() and the CACHE_config() function through the Properties page.

To access the Properties page, right-click on CACHE and select Properties from the drop-down menu. (see Figure 3–4)

The first time the Properties page appears, only the Open Handle to CACHE check-box can be selected.

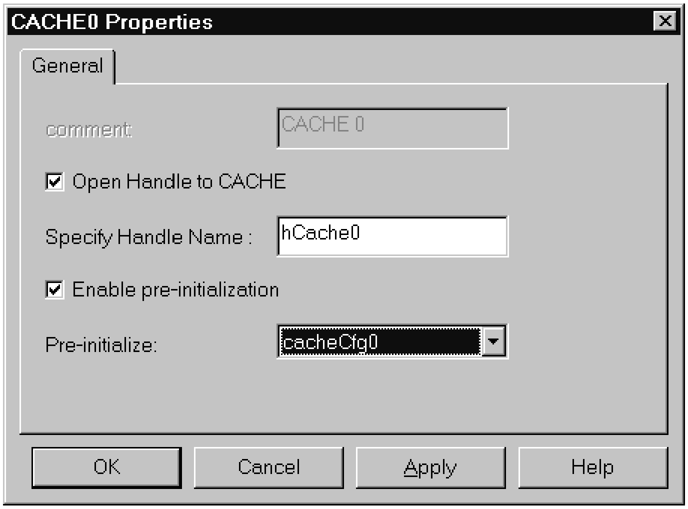
CACHE_NOTHING is used to indicate that there is no configuration object selected for this serial port.

To pre-initialize a CACHE port, check the Enable pre-initialization box. One of the available configuration objects (see Section 3.6.2, *CACHE Configuration Manager*, on page 3-23) can then be selected for this channel through the Pre-initialize drop-down list..

If CACHE_NOTHING remains selected, no configuration object will be generated for the related CACHE handle. (see Section 3.6.4, *C Code Generation for CACHE Module*, on page 3-27).

In Figure 3–4, the Open Handle to CACHE option is checked and the handle object hCACHE0 is now accessible (renaming allowed). The CACHE_open() function is now generated with hCACHE0 containing the return handle address.

Figure 3–4. CACHE Properties Page With Handle Object Accessible



3.2.4 C Code Generation for CACHE Module

Two C files are generated from the configuration tool:

- Header file
- Source file

3.2.4.1 Header File

The header file includes all the csl header files of the modules and contains the CACHE handle and configuration objects defined by the configuration tool (see Example 3–1).

Example 3–1. CACHE Header File

```
extern CACHE_Config cacheCfg0;
extern CACHE_Handle hCache0;
```

3.2.4.2 Source File

The source file includes the declaration of the handle object and the configuration structures (see Example 3–2).

Example 3–2. CACHE Source File (Declaration Section)

```
/*  Config Structures */
CACHE_Config cacheCfg0 = {
    0x0004,        /*  Global Control Register   */
    0x0000,        /*  N–Way Control Register   */
    0x000c,        /*  1/2 Ram–set 1 Control Register   */
    0x0000,        /*  1/2 Ram–set 1 Tag Register   */
    0x000c,        /*  1/2 Ram–set 2 Control Register   */
    0x0000         /*  1/2 Ram–set 2 Tag Register   */
};

/*  Handles  */
CACHE_Handle hCache0;
```

The source file contains the Handle and Configuration Pre-Initialization using the CSL CACHE API functions, CACHE_open() and CACHE_config() (see Example 3–3).

These two functions are encapsulated in a unique function, CSL_cfgInit(), which is called from your main C file. CACHE_open() and CACHE_config() are generated only if Open Handle to CACHE and Enable pre-initialization (with a selected configuration other than CACHE_NOTHING) are, respectively, checked under the CACHE Resource Manager Properties page.

Example 3–3. CACHE Source File (Body Section)

```
void CSL_cfgInit()
{
    CSL_init();
    hCache0 = CACHE_open(0xFFFF);
    CACHE_config(hCache0, &cacheCfg0);
}
```

3.3 DMA Module

3.3.1 Overview

The DMA module facilitates configuration of the Direct Memory Access (DMA) controller. The DMA module consists of a configuration manager and a resource manager.

The configuration manager allows creation of an object that contains the complete set of register values needed to configure a DMA channel. The resource manager associates a configuration object with a specific DMA channel.

Figure 3–5 illustrates the DMA sections menu on the CSL graphical user interface (GUI).

Figure 3–5. DMA Sections Menu

The DMA includes the following sections:

- **DMA Configuration Manager**: Allows you to create configuration objects by setting the peripheral registers related to the DMA.
- **DMA Resource Manager**: Allows you to select a DMA channel and to associate a configuration object to this channel . The six channel handle objects are predefined.

3.3.2 DMA Configuration Manager

The DMA Configuration Manager allows you to create DMA Channel configurations through the Properties page and to generate the configuration objects.

3.3.2.1 Creating/Inserting a configuration

There is no predefined configuration object available.

To configure a DMA channel through the Peripheral Registers, you must insert a new configuration object.

To insert a new configuration object, right-click on the DMA Configuration Manager and select insert dmaCfg from the drop-down menu. The configuration objects can be renamed. Their use depends on the on-chip device resources. Because six channels are available, a maximum of six configurations can be used simultaneously.

Note: Note: One DMA configuration may be used by more than one DMA channel.

3.3.2.2 Deleting/Renaming an Object

To delete or rename an object, right-click on the configuration object you want to delete or rename. Select Delete to delete a configuration object. Select Rename to rename the object.

If a configuration object is used by one of the predefined handle objects of the DMA Resource Manager, the Delete and Rename options are grayed out and non-usable. The Show Dependency option is accessible and shows which device is using the configuration object. See Section 2.2, *Introduction to DSP/BIOS Configuration Tool: CSL Tree,* on page 2-3.

3.3.2.3 Configuring the Object Properties

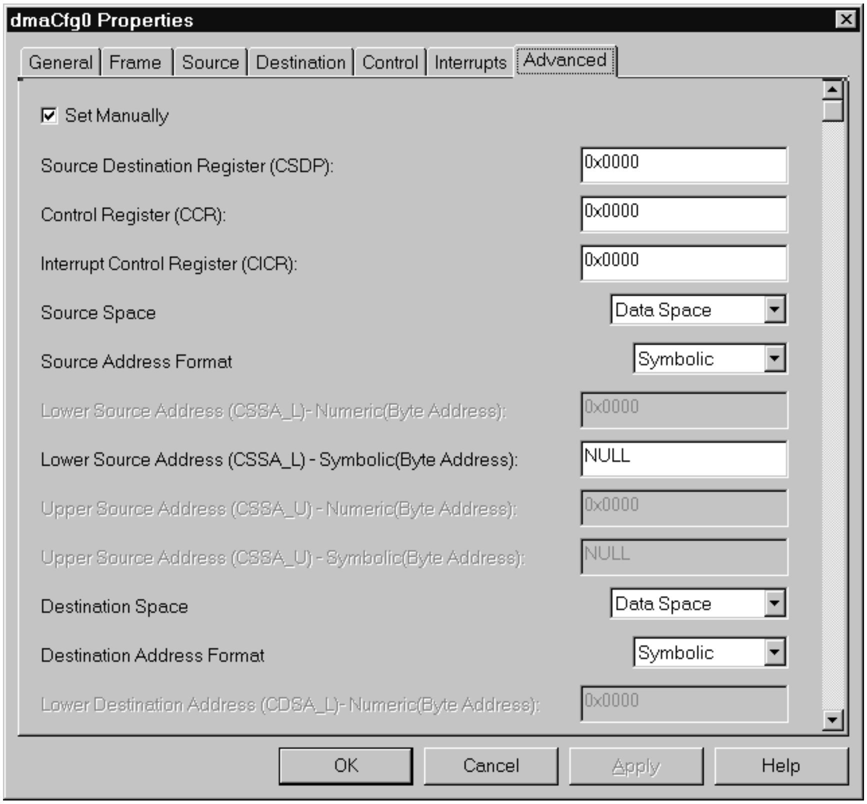
You can configure object properties through the Properties dialog box. (See Figure 3–6). To access the Properties dialog box, right-click on a configuration object and select Properties. By default, the General page of the Properties dialog box is displayed.

The Properties pages allow you to set the Peripheral registers related to the DMA. You can set the configuration options through the following Tab pages:

- ❑ Transfer Modes: Allows you to configure the Priority, Sync Events, ABU/Multi-frame
- ❑ Source/Destination: Allows you to configure the Address, Index, Element/Frame Count
- ❑ Autoinit: Allows you to configure the Reload Registers
- ❑ Advanced A and B Pages: This page contains the full hexadecimal register values and reflects the option setting of the previous pages. Also, the full register values can be entered directly and the new options are mirrored in the related pages automatically.

Figure 3–6, *DMA Properties Page*, depicts the Properties Page dialog box.

Figure 3–6. DMA Properties Page



Each Tab page is composed of several options that are set to a default value (at device reset).

3.3.2.4 Selecting the Address Formats

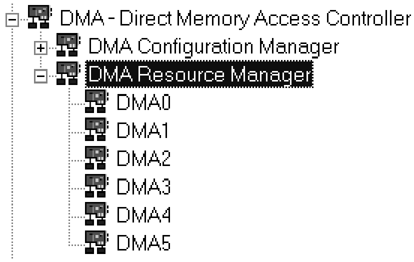
The source, destination, and addresses can be specified in either a numeric format (hard coded address) or a symbolic format. Before setting any addresses, it is suggested that you ensure that the right format is selected in the Source Address Format and Destination Address Format pull-down menus located on the Source and Destination tabs of the Properties page.

3.3.3 DMA Resource Manager

The DMA Resource Manager allows you to generate the DMA_open() and DMA_config() CSL functions.

Figure 3–7 illustrates the DMA Resource Manager menu on the CSL graphical user interface (GUI).

Figure 3–7. DMA Resource Manager Menu



3.3.3.1 Predefined Objects

The four channel handle objects are predefined and each is associated with a supported on-chip DMA channel as follows:

- **DMA0** – Default handle name: hDma0
- **DMA1** – Default handle name: hDma1
- **DMA2** – Default handle name: hDma2
- **DMA3**– Default handle name: hDma3
- **DMA4** – Default handle name: hDma4
- **DMA5** – Default handle name: hDma5

3.3.3.2 Properties Page

You can generate the DMA_open() and DMA_config() CSL functions through the Properties page.

To access the Properties page, right-click on a predefined DMA channel and select Properties from the drop-down menu (see Figure 3–8).

The first time the Properties page appears, only the Open Handle to DMA check-box can be selected. Select this to open the DMA channel, allowing pre-initialization.

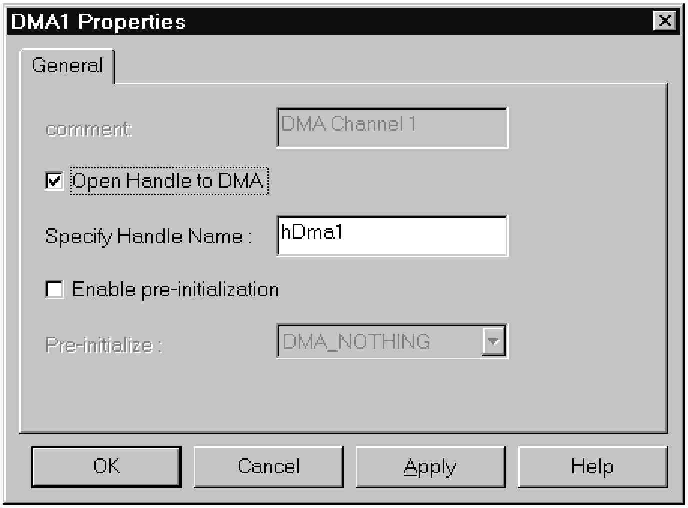
DMA_NOTHING is used to indicate that there is no configuration object selected for this DMA.

To pre-initialize the DMA channel, check the Enable pre–initialization check-box. You can then select one of the available configuration objects (see section 3.3.2 , *DMA Configuration Manager*) for this channel through the pre-initialize drop-down list.

If DMA_NOTHING is selected, no configuration object is generated for the related DMA handle. See section 3.3.4, *C Code Generation for DMA Module*, on page 3-12.

In the example shown in Figure 3–8, the Open DMA Channel option is checked and the handle object hDma1 is now accessible (The handle object can be renamed by typing the new name in the box provided). The DMA_open() function is now generated with hDma1 containing the returned handle address.

Figure 3–8. DMA Properties Page With Handle Object Accessible



3.3.4 C Code Generation for DMA Module

Two C files are generated from the configuration tool:

- ❑ Header file
- ❑ Source file.

3.3.4.1 Header File

The header file includes all the csl header files of the modules and contains the DMA handles and configuration objects generated by the configuration tool (see Example 3–4).

Example 3–4. DMA Header File

```
extern DMA_Config dmaCfg0;
extern DMA_Handle hDma1;
```

3.3.4.2 Source File

The source file includes the declaration of the channel handle objects and the configuration structures.

Example 3–5. DMA Source File (Declaration Section)

```
/*  Config Structures */
DMA_Config dmaCfg0 = {
    0x0000,        /*  Source Destination Register (CSDP)  */
    0x0000,        /*  Control Register (CCR)  */
    0x0000,        /*  Interrupt Control Register (CICR)  */
    NULL,        /*  Lower Source Address (CSSA_L) – Symbolic  */
    NULL,        /*  Upper Source Address (CSSA_U) – Symbolic  */
    NULL,        /*  Lower Destination Address (CDSA_L) – Symbolic  */
    NULL,        /*  Upper Destination Address (CDSA_U) – Symbolic  */
    0x0001,        /*  Element Number (CEN)  */
    0x0001,        /*  Frame Number (CFN)  */
    0x0000,        /*  Frame Index (CFI)  */
    0x0000         /*  Element Index (CEI)  */
};

/*  Handles  */
DMA_Handle hDma1;
```

The source file contains the Handle and Configuration Pre-Initialization using the CSL DMA API functions, DMA_open() and DMA_config() (see Example 3–6).

These two functions are encapsulated in a unique function, CSL_cfgInit(), which is called from your main C file. DMA_open() and DMA_config() are generated only if Open Handle to DMA and Enable pre-initialization (with a selected configuration other than DMA_NOTHING) are, respectively, checked under the DMA Resource Manager Properties page.

Example 3–6. DMA Source File (Body Section)

```
void CSL_cfgInit()
{
    CSL_init();
    hDma1 = DMA_open(DMA_CHA1, DMA_OPEN_RESET);
    DMA_config(hDma1, &dmaCfg0);
}
```

3.4 EMIF Module

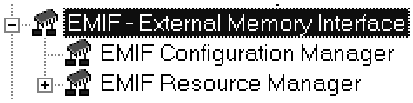
3.4.1 Overview

The EMIF module facilitates configuration of the External Memory Interface (EMIF). The EMIF module consists of a configuration manager and a resource manager.

The configuration manager allows creation of an object that contains the complete set of register values needed to configure a EMIF channel. The resource manager associates a configuration object with a specific EMIF channel.

Figure 3–9 illustrates the EMIF sections menu on the CSL graphical user interface (GUI).

Figure 3–9. EMIF Sections Menu



The EMIF includes the following two sections:

- **EMIF Configuration Manager**: Allows you to create configuration objects by setting the Peripheral registers related to the EMIF.
- **EMIF Resource Manager**: Allows you to associate a pre-configuration object to the EMIF.

3.4.2 EMIF Configuration Manager

The EMIF Configuration Manager allows you to create EMIF configurations through the Properties page and generate the configuration objects.

3.4.2.1 Creating/Inserting a Configuration Object

There is no predefined configuration object available.

To configure a EMIF channel through the Peripheral Registers, you must insert a new configuration object.

To insert a new configuration object, right-click on the EMIF Configuration Manager and select insert emifCfg from the drop-down menu. The configuration objects can be renamed. Their use depends on the on-chip device resources. Because only one channel is available, a maximum of one configuration can be used simultaneously.

3.4.2.2 Delete/Rename Object

To delete or rename an object, right-click on the configuration object you want to delete or rename. Select Delete to delete a configuration object. Select Rename to rename the object.

If a configuration object is used by one of the predefined handle objects of the EMIF Resource Manager (see section 3.4.3, *EMIF Resource Manager*, on page 3-16), the Delete and Rename options are grayed out and non-usable. The Show Dependency option is accessible and shows which device is using the configuration object. See section 2.2, *Introduction to DSP/BIOS Configuration Tool: CSL Tree,* on page 2-3.

3.4.2.3 Configuring the Object Properties

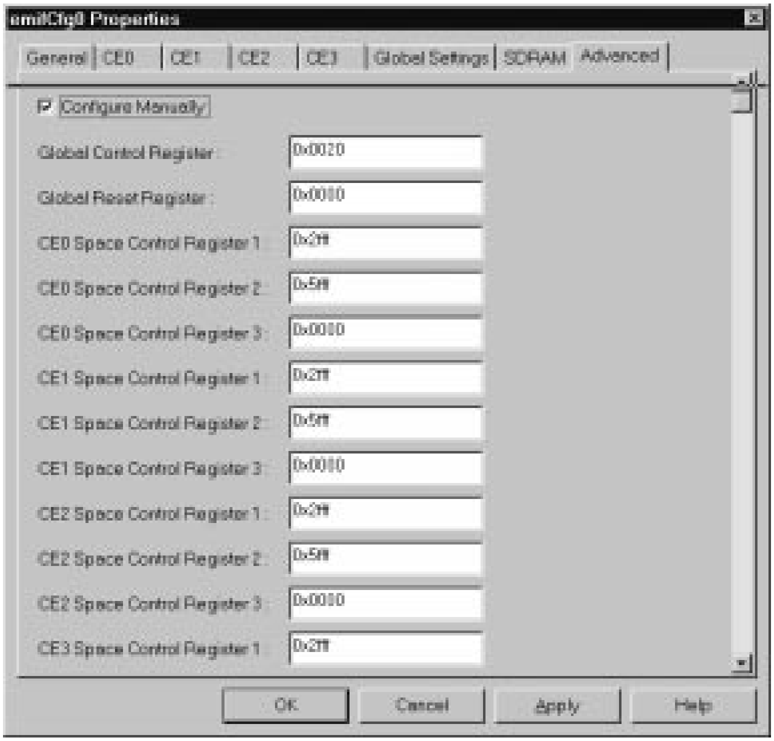
You can configure object properties through the Properties dialog box. (See Figure 3–10). To access the Properties dialog box, right-click on a configuration object and select Properties. By default, the General page of the Properties dialog box is displayed.

The Properties pages allow you to set the Peripheral registers related to the DMA. You can set the configuration options through the following Tab pages:

- ❑ Global Settings: Allows you to configure Hold, clock enable, clock frequency
- ❑ CE0 : Allows you to configure CE0 memory space
- ❑ CE1 : Allows you to configure configuration of CE1 memory space
- ❑ CE2 : Allows you to configure CE2 memory space
- ❑ CE3 : Allows you to configure CE3 memory space
- ❑ SDRAM: Allows you to configure SDRAM settings
- ❑ Advanced Page: This page contains the full hexadecimal register values and reflects the option setting of the previous pages. Also, the full register values can be entered directly and the new options are mirrored in the related pages automatically.

Figure 3–10, *EMIF Properties Page*, depicts the Properties Page dialog box.

Figure 3–10. EMIF Properties Page



Each Tab page is composed of several options that are set to a default value (at device reset).

The options represent the fields of the EMIF registers; the associated field name is shown in parenthesis. For further details on the fields and registers, refer to the *EMIF Module* chapter in the *TMS320C55x Chip Support Library API Reference Guide* (literature number SPRU433).

3.4.3 EMIF Resource Manager

The EMIF Resource Manager allows you to generate the EMIF_config() CSL function with the predefined configuration as parameter. Because only one EMIF is supported, only one resource is available and used as the default.

3.4.3.1 Properties Page

You can generate the EMIF_open() and EMIF_config() CSL functions through the Properties page.

To access the Properties page, right-click on a predefined EMIF channel and select Properties from the drop-down menu (see Figure 3–11).

The first time the Properties page appears, only the Open Handle to EMIF check-box can be selected. Select this to open the EMIF channel, allowing pre-initialization.

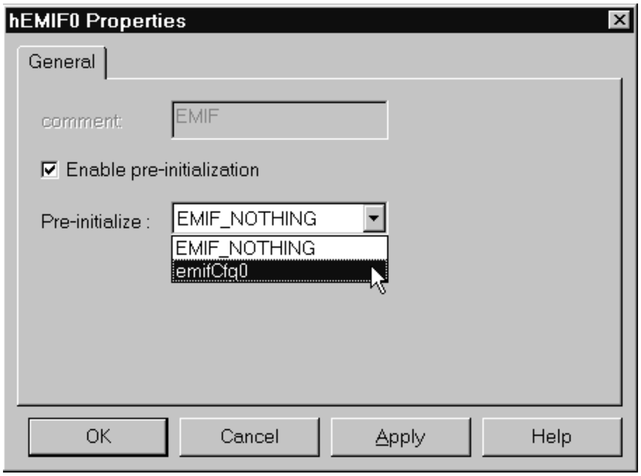
EMIF_NOTHING is used to indicate that there is no configuration object selected for this EMIF.

To pre-initialize the EMIF channel, check the Enable pre–initialization check-box. You can then select one of the available configuration objects(see section 3.4.2 , EMIF *Configuration Manager*) for this channel through the pre-initialize drop-down list.

If EMIF_NOTHING is selected, no configuration object is generated for the related EMIF handle. See section 3.4.4, *C Code Generation for EMIF Module, on page 3-12*.

In the example shown in Figure 3–11, the Open EMIF Channel option is checked and the handle object hEmif1 is now accessible (The handle object can be renamed by typing the new name in the box provided). The EMIF_open() function is now generated with hDma1 containing the returned handle address.

Figure 3–11. EMIF Resource Manager Dialog Box



3.4.4 C Code Generation for EMIF Module

Two C files are generated from the configuration tool:

- ❑ Header file
- ❑ Source file.

3.4.4.1 Header File

The header file includes all the csl header files of the modules and contains the EMIF handles, and configuration objects generated by the configuration tool (see Example 3–7).

Example 3–7. EMIF Header File

```
extern EMIF_Config emifCfg0;
```

3.4.4.2 Source File

The source file includes the declaration of the configuration structures (see Example 3–8).

Example 3–8. EMIF Source File (Declaration Section)

```
/*  Config Structures */
EMIF_Config emifCfg0 = {
    0x0020,        /*  Global Control Register   */
    0x0000,        /*  Global Reset Register   */
    0x2fff,        /*  CE0 Space Control Register 1   */
    0x5fff,        /*  CE0 Space Control Register 2   */
    0x5fff,        /*  CE0 Space Control Register 3   */
    0x2fff,        /*  CE1 Space Control Register 1   */
    0x5fff,        /*  CE1 Space Control Register 2   */
    0x5fff,        /*  CE1 Space Control Register 3   */
    0x2fff,        /*  CE2 Space Control Register 1   */
    0x5fff,        /*  CE2 Space Control Register 2   */
    0x5fff,        /*  CE2 Space Control Register 3   */
    0x2fff,        /*  CE3 Space Control Register 1   */
    0x5fff,        /*  CE3 Space Control Register 2   */
    0x5fff,        /*  CE3 Space Control Register 3   */
    0xf948,        /*  SDRAM Control Register 1   */
    0x0080,        /*  SDRAM Period Register   */
    0x0000,        /*  SDRAM Initialization Register   */
    0x03ff         /*  SDRAM Control Register 2   */
};
```

The source file contains the Handle and Configuration Pre-Initialization using the CSL EMIF API functions, EMIF_open() and EMIF_config() (see Example 3–9).

These two functions are encapsulated in a unique function, CSL_cfgInit(), which is called from your main C file. EMIF_open() and EMIF_config() are generated only if Open Handle to EMIF and Enable pre-initialization (with a selected configuration other than EMIF_NOTHING) are, respectively, checked under the EMIF Resource Manager Properties page.

Example 3–9. EMIF Source File (Body Section)

```
void CSL_cfgInit()
{
    CSL_init();

    EMIF_config(&emifCfg0);
}
```

3.5 GPIO Module

3.5.1 Overview

The GPIO module facilitates configuration/control of the General Purpose I/O on the C55x. The module consists of a configuration manager. The configuration manager allows you to configure the directions of either the input or output of the GPIO pins.

Figure 3–12 illustrates the GPIO sections menu on the CSL graphical user interface (GUI)

Figure 3–12. GPIO Sections Menu

The Non-Multiplexed GPIO includes the following section:

- **Non-Multiplexed GPIO Configuration Manager**: Allows you to configure the GPIO Pin directions.

3.5.2 Non-Multiplexed GPIO Configuration Manager

The Non-Multiplexed GPIO Configuration Manager allows you to configure the GPIO Pin directions.

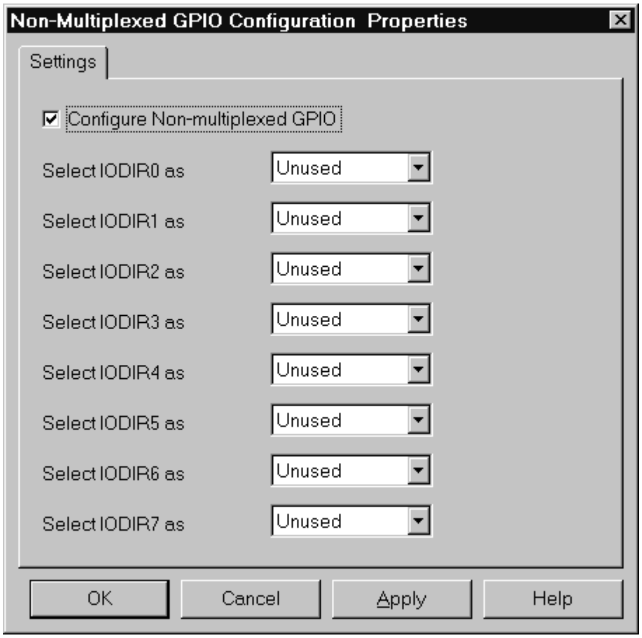
3.5.2.1 Properties Pages of the Non-Multiplexed GPIO Configuration

The Properties pages allow you to set the Peripheral registers related to the GPIO. The configuration options are divided into the following Tab page:

- Settings: Allows you to configure the Input/Output settings of GPIO Pins.

Figure 3–13, Non-Multiplexed GPIO *Properties Page*, depicts the Properties Page dialog box.

Figure 3–13. GPIO Properties Page



The settings Tab is composed of several options that are set to a default value (at device reset).

The options represent the fields of the GPIO register direction; the associated field name is shown in parenthesis. For further details of the fields and registers, refer to the *GPIO Section* chapter of the *TMS320C55x DSP CPU and Peripherals References Set* (literature number SPRU304).

3.5.3 C Code Generation for GPIO Module

Two C files are generated from the configuration tool:

- ❑ Header file
- ❑ Source file.

3.5.3.1 Header File

The header file includes all the csl header files of the modules.

3.5.3.2 Source File

The source file contains the GPIO Register set macro invocation. This macro invocation is encapsulated in a unique function, CSL_cfgInit(), which is called from your main C file.

GPIO_RSET() will be generated only if Configure Non–Multiplexed GPIO is checked under the Non-multiplexed GPIO Configuration Properties page. See Figure 3–13.

Example 3–10. GPIO Source File (Body Section)

```
void CSL_cfgInit()
{
    CSL_init();

    GPIO_RSET(IODIR, 3840);
}
```

3.6 MCBSP Module

3.6.1 Overview

The MCBSP module facilitates configuration/control of the Multi Channel Buffered Serial Port (MCBSP). The module consists of a configuration manager and a resource manager. The configuration manager allows creation of one or more configuration objects. The configuration objects contain all of the data necessary to set the MCBSP Control Registers. The resource manager associates a configuration object with a specified port.

Figure 3–14 illustrates the GPIO sections menu on the CSL graphical user interface (GUI)

Figure 3–14. MCBSP Sections Menu

The MCBSP includes the following two sections:

- **MCBSP Configuration Manager**: Allows you to create configuration objects. No predefined configuration objects.
- **MCBSP Resource Manager**: Allows you to select a device and to associate a configuration object to that device. Three handle objects are predefined.

3.6.2 MCBSP Configuration Manager

The MCBSP Configuration Manager allows you to create device configurations through the Properties page and to generate the configuration objects.

3.6.2.1 Creating/Inserting a Configuration Object

There is no predefined configuration object available.

To configure a MCBSP port through the peripheral registers, you must insert a new configuration object.

To insert a new configuration object, right-click on the MCBSP Configuration Manager and select insert mcbspCfg from the drop-down menu. The configuration objects can be renamed. Their use depends upon the on-chip device resources.

Note: Note: The number of configuration objects is unlimited. Several configurations can be created and the user can select the right one for a specific port and can change the configuration later just by selecting a new one under the MCBSP Resource Manager. The goal is to provide more flexibility and to reduce the time required to modify register values.

3.6.2.2 Deleting/Renaming an Object

To delete or to rename an object, right-click on the configuration object you want to delete or rename. Select Delete to delete a configuration object. Select Rename to rename the object.

If a configuration object is used by one of the predefined handle objects of the MCBSP Resource Manager, the Delete and Rename options are grayed out and non-usable. The Show Dependency option is accessible and shows which device is using the configuration object. See Section 2.2, *Introduction to DSP/BIOS Configuration Tool: CSL Tree*, on page 2-3.

3.6.2.3 Configuring the Object Properties

The Properties pages allow you to set the Peripheral registers related to the MCBSP Port (see Figure 3–15). To access the Properties dialog box, right-click on a configuration object and select Properties. By default, the General page of the Properties dialog box is displayed.

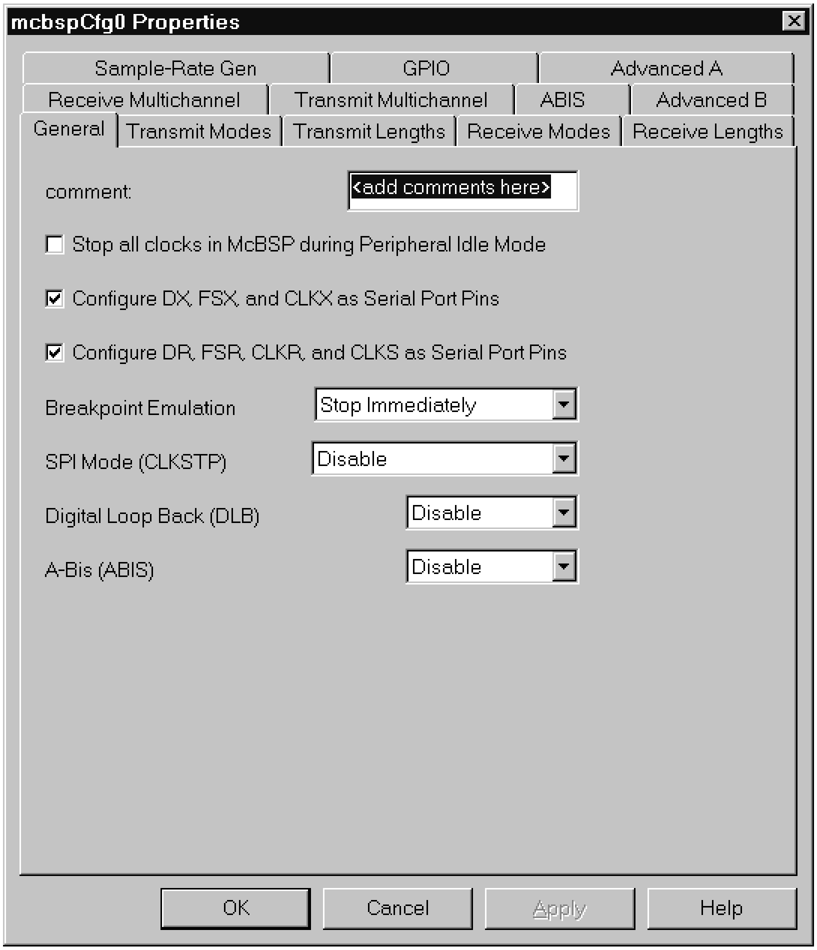
The Properties pages allow you to set the Peripheral registers related to the MCBSP. you can set the configuration options through the following pages:

- General: Allows you to configure the Digital Loopback, ABIS Mode, Breakpoint Emulation.
- Transmit Modes: Allows you to configure the Interrupt mode, Frame Sync, Clock control.
- Transmit Lengths: Allows you to configure the Phase, elements-per-word, elements per frame.
- Receiver Modes: Allows you to configure the Interrupt mode, Frame Sync, Clock control.
- Receiver Lengths: Allows you to configure the Phase, elements-per-word, elements per frame.
- Sample-Rate Generator: Allows you to configure the Sample-Rate Generator (Frame Setup).
- Receive Multi-channel: Allows you to configure the Element and Block partitioning.
- Transmit Multi–channel: Allows you to configure the Element and Block partitioning.
- Some fields are activated according to the setup of the Transmitter, Receiver, and Sample-rate generator options.

- ❑ Advanced A and B: Summary of the previous pages. This page contains the full hexadecimal register values and reflects the setting of the options done under the previous pages
- ❑ The full register values can be entered directly and the new options will be mirrored on the corresponding pages automatically.

Figure 3–15, *MCBSP Properties Page*, depicts the Properties Page.

Figure 3–15. MCBSP Properties Page



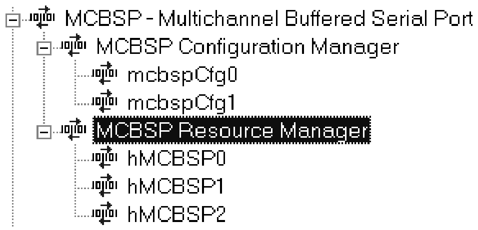
Each Tab page is composed of several options that are set to a default value (at device reset).

3.6.3 MCBSP Resource Manager

The MCBSP Resource Manager allows you to generate the MCBSP_open() and the MCBSP_config() CSL functions.

Figure 3–16 illustrates the MCBSP Resource Manager menu on the CSL graphical user interface (GUI).

Figure 3–16. MCBSP Resource Manager Menu



3.6.3.1 Predefined Objects

Three handle objects are predefined and each of them is associated with a supported on-chip MCBSP port.

- **MCBSP0** – Default handle name: hMcbsp0
- **MCBSP1** – Default handle name: hMcbsp1
- **MCBSP2** – Default handle name: hMcbsp2

Note: The above objects can neither be deleted nor renamed.

A configuration can be enabled if at least one configuration object was defined previously. See Section 3.6.2, *MCBSP Configuration Manager*, on page 3-23.

3.6.3.2 Properties Page

You can generate the MCBSP_open() and MCBSP_config() CSL functions through the Properties page.

To access the Properties page, right-click on a predefined MCBSP channel and select Properties from the drop-down menu (see Figure 3–17).

The first time the Properties page appears, only the Open Handle to MCBSP check-box can be selected. Select this to open the MCBSP channel, allowing pre-initialization.

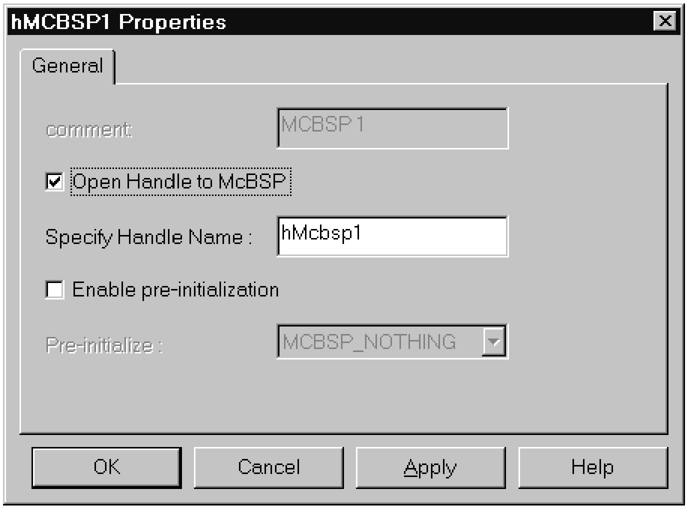
MCBSP_NOTHING is used to indicate that there is no configuration object selected for this serial port.

To pre-initialize a MCBSP port, check the Enable Pre-Initialization box. You can then select one of the available configuration objects (see Section 3.6.2, *MCBSP Configuration Manager*, on page 3-23) for this channel through the pre-initialize drop-down list..

If MCBSP_NOTHING is selected, no configuration object is generated for the related MCBSP handle. (see Section 3.6.4, *C Code Generation for MCBSP Module*, on page 3-27).

In the example shown in Figure 3–17, the Open Handle to MCBSP option is checked and the handle object hMcbsp1 is now accessible (The handle object can be renamed by typing the new name in the box provided). The MCBSP_open() function is now generated with hMcbsp0 containing the returned handle address.

Figure 3–17. MCBSP Properties Page With Handle Object Accessible



3.6.4 C Code Generation for MCBSP Module

Two C files are generated from the configuration tool:

- ❑ Header file
- ❑ Source file.

3.6.4.1 Header File

The header file includes all the csl header files of the modules and contains the MCBSP handle and configuration objects defined from the configuration tool (see Example 3–11).

Example 3–11. MCBSP Header File

```
extern MCBSP_Config mcbsCfg0;
extern MCBSP_Handle hMcbsp1;
```

3.6.4.2 Source File

The source file includes the declaration of the handle object and the configuration structures (see Example 3–12).

Example 3–12. MCBSP Source File (Declaration Section)

```
/*  Config Structures */
MCBSP_Config mcbspCfg0 = {
    0x0000,        /*  Serial Port Control Register 1   */
    0x0000,        /*  Serial Port Control Register 2   */
    0x0000,        /*  Receive Control Register 1   */
    0x0000,        /*  Receive Control Register 2   */
    0x0000,        /*  Transmit Control Register 1   */
    0x0000,        /*  Transmit Control Register 2   */
    0x0000,        /*  Sample Rate Generator Register 1   */
    0x0000,        /*  Sample Rate Generator Register 2   */
    0x0000,        /*  Multi–channel Control Register 1   */
    0x0000,        /*  Multi–channel Control Register 2   */
    0x0000,        /*  Pin Control Register   */
    0x0000,        /*  Receive Channel Enable Register Partition A   */
    0x0000,        /*  Receive Channel Enable Register Partition B   */
    0x0000,        /*  Receive Channel Enable Register Partition C   */
    0x0000,        /*  Receive Channel Enable Register Partition D   */
    0x0000,        /*  Receive Channel Enable Register Partition E   */
    0x0000,        /*  Receive Channel Enable Register Partition F   */
    0x0000,        /*  Receive Channel Enable Register Partition G   */
    0x0000,        /*  Receive Channel Enable Register Partition H   */
    0x0000,        /*  Transmit Channel Enable Register Partition A   */
    0x0000,        /*  Transmit Channel Enable Register Partition B   */
    0x0000,        /*  Transmit Channel Enable Register Partition C   */
    0x0000,        /*  Transmit Channel Enable Register Partition D   */
    0x0000,        /*  Transmit Channel Enable Register Partition E   */
    0x0000,        /*  Transmit Channel Enable Register Partition F   */
    0x0000,        /*  Transmit Channel Enable Register Partition G   */
    0x0000         /*  Transmit Channel Enable Register Partition H   */
};

/*  Handles  */
MCBSP_Handle hMcbsp1;
```

The source file contains the Handle and Configuration Pre-Initialization using the CSL MCBSP API functions, MCBSP_open() and MCBSP_config() (see Example 3–13).

These two functions are encapsulated in a unique function, CSL_cfgInit(), which is called from your main C file. MCBSP_open() and MCBSP_config() are generated only if Open Handle to DMA and Enable pre-initialization (with a selected configuration other than MCBSP_NOTHING) are, respectively, checked under the MCBSP Resource Manager Properties page.

Example 3–13. MCBSP Source File (Body Section)

```
void CSL_cfgInit()
{
    CSL_init();

    hMcbsp1 = MCBSP_open(MCBSP_PORT1, MCBSP_OPEN_RESET);
    MCBSP_config(hMcbsp1, &mcbspCfg0);
}
```

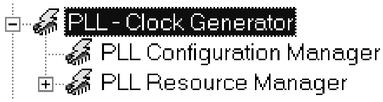
3.7 PLL Module

3.7.1 Overview

The PLL module facilitates programming of the Phase Locked Loop controlling C55xx clock. The PLL module consists of a configuration manager and a resource manager. The configuration manager allows creation of one or more configuration objects. A configuration object consists of the necessary register settings to control the PLL. The resource manager associates a selected configuration with the PLL.

Figure 3–18 illustrates the PLL sections menu on the CSL graphical user interface (GUI).

Figure 3–18. PLL Sections Menu



The PLL includes the following two sections:

❑ **PLL Configuration Manager**: Allows you to create configuration objects by setting the Peripheral registers related to the PLL.

❑ **PLL Resource Manager**: Allows you to associate a pre-configuration object to the PLL.

3.7.2 PLL Configuration Manager

The PLL Configuration Manager allows you to create PLL configurations through the Properties page and to generate the configuration objects.

3.7.2.1 Creating/Inserting a configuration

There is no predefined configuration object.

To configure a PLL setting through the Peripheral Registers, you must insert a new configuration object.

To insert a new configuration object, right-click on the PLL Configuration Manager and select Insert pllCfg. The configuration objects can be renamed.

Note: Note: The number of configuration objects is unlimited. Several configurations can be created. You can select one for the PLL and can change the configuration later just by selecting another configuration under the PLL Resource Manager. This feature allows you more flexibility and reduces the time required to modify register values.

3.7.2.2 Deleting/Renaming and Object

To delete or rename an object, right-click on the configuration object you want to delete or rename. Select Delete to delete a configuration object. Selet Rename to rename the object.

If a configuration object is used by one of the predefined handle objects of the PLL Resource Manager, the Delete and Rename options are grayed out and non-usable. The Show Dependency option is accessible and shows which device is using the configuration object. See Section 2.2, *Introduction to DSP/BIOS Configuration Tool: CSL Tree*, on page 2-3.

3.7.2.3 Configuring the Object Properties

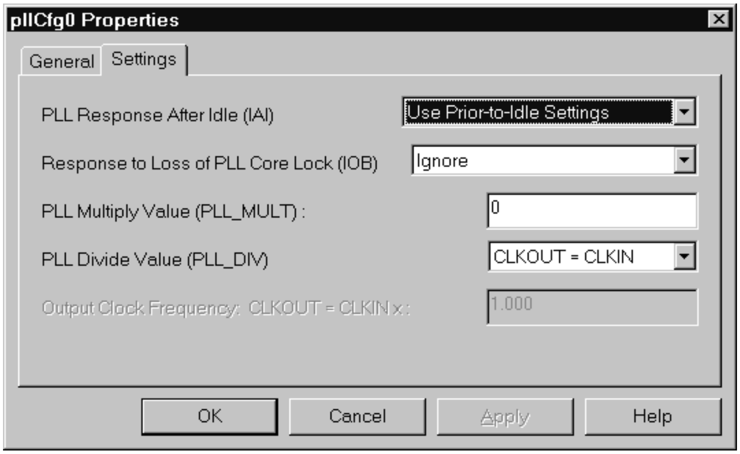
You can configure object properties through the Properties dialog box (see Figure 3–19). To access the Properties dialog box, right-click on a configuration object and select Properties. By default, the General page of the Properties dialog box is displayed.

The Properties pages allow you to set the Peripheral registers related to the PLL. You can set the configuration options through the following tab page:

- ❑ Settings: Allows you to configure the Counter Value, Multiplier, Divide Factor

Figure 3–19, *PLL Properties Page*, depicts the Properties Page dialog box.

Figure 3–19. PLL Properties Page



Each Tab page is composed of several options that are set to a default value (at device reset).

The options represent the fields of the PLL registers; the associated field name is shown in parenthesis. For further details of the fields and registers, refer to the *Expansion Bus* chapter of the *TMS320C55xx Chip Support Library API Reference Guide* (literature number SPRU433).

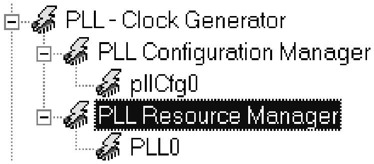
3.7.3 PLL Resource Manager

The PLL Resource Manager allows you to generate the PLL_config() CSL function.

Because only one PLL is supported, only one resource is available and used as the default.

Figure 3–20 illustrates the PLL Resource Manager menu on the CSL graphical user interface (GUI).

Figure 3–20. PLL Resource Manager Menu



3.7.3.1 Properties Page

You can generate the PLL_config() CSL function through the Properties page.

To access the Properties page, right-click on a predefined PLL channel and select Properties from the drop-down menu (see Figure 3–21).

The first time the properties page appears, only the Enable Configuration PLL check box can be selected. Select this to enable the PLL configuration.

The Properties page is accessible by right-clicking on the drop-down menu option Properties.

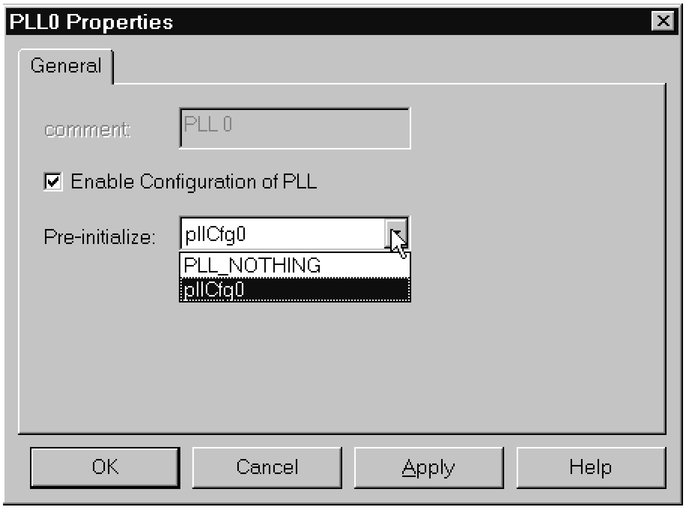
PLL_NOTHING is used to indicate that there is no configuration object selected for the PLL.

To pre-initialize the PLL, check the Enable Configuration of PLL box. One of the available configuration objects(see Section 3.3.2 , *PLL Configuration Manager*) can then be selected for the PLL channel through the Pre–Initialize drop-down list.

If PLL_NOTHING remains selected, the PLL_config() function will not be generated for the PLL.

In Figure 3–21, the pllCfg0 is selected. The PLL_config function will now be generated with hPll0 containing the return handle address.

Figure 3–21. PLL Properties Page



3.7.4 C Code Generation for PLL Module

Two C files are generated from the configuration tool:

- ❑ Header file
- ❑ Source file.

3.7.4.1 Header File

The header file includes all the csl header files of the modules and contains the PLL configuration objects defined from the configuration tool (see Example 3–14).

Example 3–14. PLL Header File

```
extern PLL_Config pllCfg0;
```

3.7.4.2 Source File

The source file includes the declaration of the configuration structures (values of the peripheral registers) (see Example 3–15).

Example 3–15. PLL Source File (Declaration Section)

```
/*  Config Structures */
PLL_Config pllCfg0 = {
    0x0000,        /*  PLL Response After Idle (IAI)   */
    0x0000,        /*  Response to Loss of PLL Core Lock (IOB)   */
    0x0000,        /*  PLL Multiply Value (PLL_MULTI)   */
    0x0000         /*  PLL Divide Value (PLL_DIV)   */
};
```

The source file contains the Pre-Initialization PLL API function PLL_config(). This function is encapsulated in a unique function, CSL_cfgInit(), which is called from your main C file (see Example 3–16).

PLL_config() is generated only if Enable Configuration of PLL is checked under the PLL Resource Manager Properties page (with a selected configuration other than PLL_NOTHING) (see Example 3–16, on page 3-33).

Example 3–16. PLL Source File (Body Section)

```
void CSL_cfgInit()
{
    CSL_init();

    PLL_config(&pllCfg0);
}
```

3.8 TIMER Module

3.8.1 Overview

The Timer module facilitates configuration/control of the on-chip Timer. The timer module consists of a configuration manager and a resource manager. The configuration manager allows the creation of one or more configuration objects. The configuration object consists of the necessary data to set the Timer control registers. The resource manager associates a selected configuration with a timer.

Figure 3–22 illustrates the Timer sections menu on the CSL graphical user interface (GUI).

Figure 3–22. Timer Sections Menu

The TIMER includes the following two sections:

- **TIMER Configuration Manager**: Allows you to create configuration objects. There are no predefined configuration objects.
- **TIMER Resource Manager**: Allows you to select a device that will be used and to associate a configuration object with that device. Three handle objects are predefined.

3.8.2 TIMER Configuration Manager

The TIMER Configuration Manager allows you to create device configurations through the Properties page and generate the configuration objects.

3.8.2.1 Creating/Inserting a configuration

There are no predefined configuration objects available.

To configure a TIMER device through the peripheral, you must insert a new configuration object.

To insert a new configuration object, right-click on the TIMER Configuration Manager and select Insert timerCfg from the drop-down menu. The configuration objects can be renamed. Their use depends on the on-chip device resources.

Note: Note: The number of configuration objects is unlimited. Several configurations can be created and you can select the right one for a specific device and change the configuration later just by selecting a new one under the TIMER Resource Manager. This feature provides you with more flexibility and reduces the time required to modify register values.

3.8.2.2 Deleting/Renaming an Object

To delete or to rename an object, right-click on the configuration object you want to delete or rename. Select Delete to delete a configuration object. Select Rename to rename the object.

If a configuration object is used by one of the predefined handle objects of the TIMER Resource Manager, the Delete and Rename options are grayed out and non-usable. The Show Dependency option is accessible and shows which device is using the configuration object. See Section 2.2, *Introduction to DSP/BIOS Configuration Tool: CSL Tree*, on page 2-3.

3.8.2.3 Configuring the Object Properties

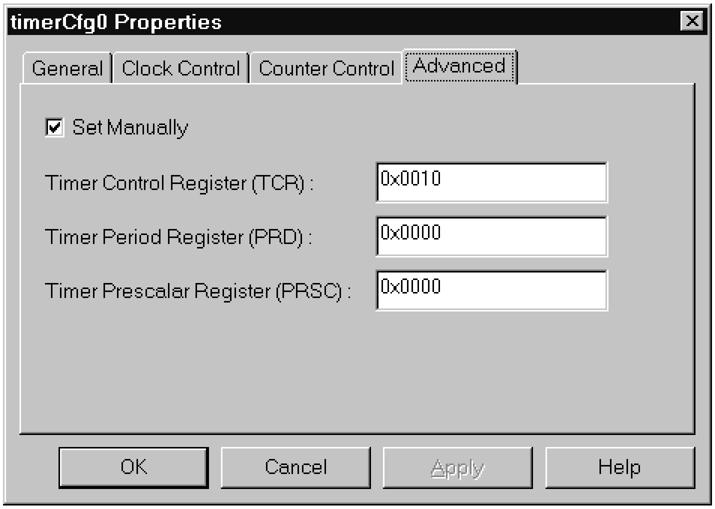
You can configure object properties through the Properties dialog box (see Figure 3–23). To access the Properties dialog box, right-click on a configuration object and select Properties. By default, the General page of the Properties dialog box is displayed.

The Properties pages allow you to set the Peripheral registers related to the TIMER. You can set the configuration options through the following tab pages:

- General: Allows you to configure the Breakpoint Emulation
- Counter Control: Allows you to configure the Counter configuration
- Advanced Page: Allows you to configure the Summary of the previous three pages
- This page contains the full hexadecimal register values and reflects the setting of the three pages
- The full register values can be entered directly and the new options will be mirrored on the previous three pages automatically

Figure 3–23, *TIMER Properties Page*, depicts the Properties Page dialog box.

Figure 3–23. TIMER Properties Page



Each Tab page is composed of several options that are set to a default value (at device reset).

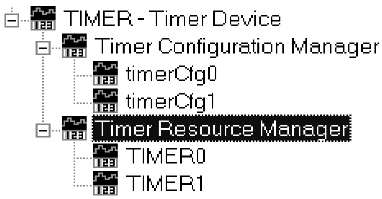
The options represent the fields of the TIMER registers; the associated field name is shown in parenthesis. For further details on the fields and registers, refer to the *Timers* chapter in the *TMS320C55xx Chip Support Library API Reference Guide* (SPRU433).

3.8.3 TIMER Resource Manager

The TIMER Resource Manager allows you to generate the TIMER_open() and the TIMER_config() CSL functions.

Figure 3–24 illustrates the TIMER Resource Manager menu.

Figure 3–24. Timer Resource Manager Menu



3.8.3.1 Predefined Objects

Two handle objects are predefined and each of them is associated with a supported on-chip TIMER device.

- **TIMER0** – Default handle name: hTimer0
- **TIMER1** – Default handle name: hTimer1

Note: The above objects can neither be deleted nor renamed.

A configuration is enabled if at least one configuration object is defined previously in section 3.8.2, *TIMER Configuration Manager*, on page 3-35.

3.8.3.2 Properties Page

You can generate the TIMER_config and TIMER_open CSL functions through the Properties page.

To access the Properties page, right-click on a predifined TIMER handle object and select Properties from the drop-down menu (see Figure 3–25).

The first time the properties page appears, only the Open Handle to Timer check-box can be selected. Select this to open the TIMER configuration, allowing pre-initialization.

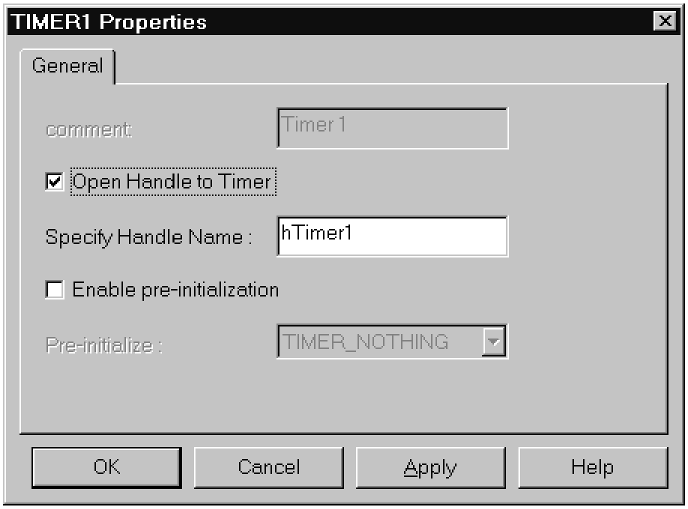
TIMER_CFGNULL is used to indicate that there is no configuration object selected for this device.

To pre-initialize the Timer, check the Enable Pre-Initialization box. One of the available configuration objects(see Section 3.3.2 , *Timer Configuration Manager*) can then be selected for this channel through the Pre–Initialize drop-down list.

If TIMER_CFGNULL is selected, no configuration object will be generated for the related TIMER handle. (See Section 3.8.4, *C Code Generation for TIMER*, on page 3-39.)

In Figure 3–25, the Open Handle to TIMER option is checked and the handle object hTimer0 is now accessible (renaming allowed). The TIMER_open() function will be generated with hTimer0 containing the return handle address.

Figure 3–25. Timer Properties Page With Handle Object Accessible



3.8.4 C Code Generation for TIMER

Two C files are generated from the configuration tool:

- Header file
- Source file.

3.8.4.1 Header File

The header file includes all the csl header files of the modules and contains the TIMER handle and configuration objects defined from the configuration tool (see Example 3–17).

Example 3–17. Timer Header File

```
extern TIMER_Config timerCfg0;
extern TIMER_Handle hTimer1
```

3.8.4.2 Source File

The source file includes the declaration of the handle object and the configuration structures (see Example 3–18).

Example 3–18. Timer Source File (Declaration Section)

```
/*  Config Structures */
TIMER_Config timerCfg0 = {
    0x0010,        /*  Timer Control Register (TCR)   */
    0x0000,        /*  Timer Period Register (PRD)   */
    0x0000         /*  Timer Prescaler Register (PRSC)   */
};


/*  Handles  */
TIMER_Handle hTimer1;
```

The source file contains the Handle and Configuration Pre-Initialization using CSL TIMER API functions TIMER_open() and TIMER_config() (see Example 3–19). These two functions are encapsulated into a unique function, CSL_cfgInit(), which is called from your main C file.

TIMER_open() and TIMER_config() will be generated only if Open Handle to TIMER and Enable-Pre-Initialization (with timerCfg0) are checked on the TIMER Resource Manager Properties page.

Example 3–19. Timer Source File (Body Section)

```
void CSL_cfgInit()
{
    CSL_init();

    hTimer1 = TIMER_open(TIMER_DEV1, TIMER_OPEN_RESET);
    TIMER_config(hTimer1, &timerCfg0);
}
```

Chapter 4

CACHE Module

This chapter details descriptions and examples of the structure, functions, and macros contained in the CACHE Module.

Topic **Page**

4.1 Overview **4-2**

4.2 Configuration Structure **4-4**

4.3 Functions **4-5**

4.4 Macros **4-8**

4.1 Overview

Table 4–1 summarizes the primary API functions. A shaded row indicates functions required to control the cache interface through the CSL.

- You can perform configuration by calling either CACHE_config(), CACHE_configArgs(), or any of the SET register macros.

  Because CACHE_config() and CACHE_configArgs() initialize all 8 control registers, macros are provided to enable efficient access to individual registers when you need to set only one or two.

  Using CACHE_config() to initialize the cache registers is the recommended approach.

- Your application can also call CACHE_flush() and CACHE_enable().

The CACHE API defines macros designed for the following primary purposes:

- The *RMK* macros create individual control-register masks for the following purposes:
  - To initialize an CACHE_Config structure that you then pass to functions such as CACHE_config().
  - To use as arguments for functions such as CACHE_configArgs().
  - To use as arguments for the appropriate SET macro.
- Other macros are available primarily to facilitate reading and writing individual bits and fields in thet imer control registers.

Table 4–1 (c) lists the most commonly used macros. Section 4.4 includes a description of all CACHE macros.

Table 4–1. CACHE Primary Summary

(a) CACHE Configuration Structure

Structure	Purpose	See page...
CACHE_Config	CACHE configuration structure used to setup the CACHE interface	4-4

(b) CACHE Functions

Function	Purpose	See page ...
CACHE_config()	Sets up CACHE using configuration structure (CACHE_Config)	4-5
CACHE_configArgs()	Sets up CACHE using register values passed to the function	4-6
CACHE_enable()	Cache Global Enable	4-7
CACHE_flush()	Cache Global Flush	4-7

(c) CACHE Macros

Macro	Purpose	See page ...
CACHE_XXXX_RMK	Creates a value to store in a particulat register (See below)	
CACHE_RSET_SET	Write a value to store in a particular register (See below)	
CACHE_RGET_GET	Read a value from a particular register (See below)	
	where XXXX is ICGC, ICFL0, ICFL1, ICWC, ICRC1, ICRTAG1, ICRC2, ICRTAG2	

4.2 Configuration Structure

This section describes the structure in the CACHE module.

CACHE_Config *CACHE configuration structure used to set up CACHE interface*

Structure CACHE_Config

Members

Uint16 icgc	ICache Global Control Register
Uint16 icfl0	ICache Flush Line Address Register 0
Uint16 icfl1	ICache Flush Line Address Register 1
Uint16 icwc	ICache N–Way Control Register
Uint16 icrc1	ICache 1/2 Ram–set 1 Control Register
Uint16 icrtag1	ICache 1/2 Ram–set 1 TAG Register
Uint16 icrc2	ICache 1/2 Ram–set 2 Control Register
Uint16 icrtag2	ICache 1/2 Ram–set 2 TAG Register

Description CACHE configuration structure used to set up the CACHE Interface. You create and initialize this structure and then pass its address to the CACHE_config() function. You can use literal values or the *CACHE_RMK* macros to create the structure member values.

Example

```
CACHE_Config Config1 = {
Uint16 icgc,
Uint16 icfl0,
Uint16 icfl1,
Uint16 icwc,
Uint16 icrc1,
Uint16 icrtag1,
Uint16 icrc2,
Uint16 icrtag2
}
```

4.3 Functions

This section describes the functions in the CACHE module.

CACHE_config *Writes value to up CACHE using configuration structure*

Function

```
void CACHE_config(
    CACHE_Config *Config
);
```

Arguments

Config   Pointer to an initialized configuration structure

Return Value

None

Description

Writes a value to up the CACHE using the configuration structure. The values of the structure are written to the port registers. See also CACHE_configArgs() and CACHE_Config.

Example

```
CACHE_Config MyConfig = {
0xFFFF,    /*icgc*/
0x00FF,    /*icfl0*/
0x00FF,    /*icfl1*/
0x000F,    /*icwc*/
0x000F,    /*icrc1 */
0x7FFF,    /*icrtag1*/
0x000F,    /*icrc2*/
0x7FFF,    /*icrtag2*/
}
CACHE_config(&MyConfig);
```

CACHE_configArgs *Writes to CACHE using register values passed to the function*

Function

void CACHE_configArgs(
Uint16 icgc,
Uint16 icfl0,
Uint16 icfl1,
Uint16 icwc,
Uint16 icrc1,
Uint16 icrtag1,
Uint16 icrc2,
Uint16 icrtag2);

Arguments

iicgc	ICache Global Control Register
icfl0	ICache Flush Line Address Register 0
icfl1	ICache Flush Line Address Register 1
icwc	ICache N–Way Control Register
icrc1	ICache 1/2 Ram–set 1 Control Register
icrtag1	ICache 1/2 Ram–set 1 TAG Register
icrc2	ICache 1/2 Ram–set 2 Control Register
icrtag2	ICache 1/2 Ram–set 2 TAG Register

Return Value

None

Description

Writes to the CACHE using the register values passed to the function. The register values are written to the CACHE registers.

You may use literal values for the arguments; or for readability, you may use the CACHE_MK macros to create the register values based on field values.

Example

```
CACHE_configArgs (
0xFFFF,    /*icgc*/
0x00FF,    /*icfl0*/
0x00FF,    /*icfl1*/
0x000F,    /*icwc*/
0x000F,    /*icrc1 */
0x7FFF,    /*icrtag1*/
0x000F,    /*icrc2*/
0x7FFF,    /*icrtag2*/
 );
```

CACHE_enable — *Enables CACHE*

Function

CACHE_enable(
    Uint16 icgcEnable
    Uint16 icwcEnable
    Uint16 icrc1Enable
    Uint16 icrc2Enable
);

Arguments

icgcEnable ICache Global Control Register
icwcEnable ICache N–Way Control Register
icrc1Enable ICache 1/4 Ram–set 1 Control Register
icrc2Enable ICache 1/4 Ram–set 2 Control Register

Return Value

None

Description

Selective enable of CACHE

Example

```
CACHE_enable(0xFFFF,0x000F,0x000F,0x000F);
```

CACHE_flush — *Flushes CACHE*

Function

CACHE_flush(
    Uint16 icgcFlush
    Uint16 icwcFlush
    Uint16 icrc1Flush
    Uint16 icrc2Flush
);

Arguments

icgcFlush ICache Global Control Register
icwcFlush ICache N–Way Control Register
icrc1Flush ICache 1/4 Ram–set 1 Control Register
icrc2Flush ICache 1/4 Ram–set 2 Control Register

Return Value

None

Description

Selective flush of instruction CACHE

Example

```
CACHE_flush(3,1,1,1);
```

4.4 Macros

CSL offers a collection of macros to gain individual access to the CACHE peripheral registers and fields..

Table 4–2 contains a list of macros available for the CACHE module. To use them, include ”csl_ebus.h”.

Table 4–2. CACHE CSL Macros Using CACHE Port Number

(a) Macros to read/write CACHE register values

Macro	Syntax
CACHE_RGET()	Uint16 CACHE_RGET(*REG*)
CACHE_RSET()	Void CACHE_RSET(*REG*, Uint16 *regval*)

(b) Macros to read/write CACHE register field values (Applicable only to registers with more than one field)

Macro	Syntax
CACHE_FGET()	Uint16 CACHE_FGET(*REG*, *FIELD*)
CACHE_FSET()	Void CACHE_FSET(*REG*, *FIELD*, Uint16 *fieldval*)

(c) Macros to create value to CACHE registers and fields (Applies only to registers with more than one field)

Macro	Syntax
CACHE_REG_RMK()	Uint16 CACHE_REG_RMK(*fieldval_n*,…*fieldval_0*) Note: *Start with field values with most significant field positions: field_n: MSB field field_0: LSB field *only writable fields allowed
CACHE_FMK()	Uint16 CACHE_FMK(*REG*, *FIELD*, *fieldval*)

(d) Macros to read a register address

Macro	Syntax
CACHE_ADDR()	Uint16 CACHE_ADDR(*REG*)

Notes:
1) *REG* indicates the register, xxx xxx.
2) *FIELD* indicates the register field name as specified in Appendix A.
   - ❑ For *REG*_FSET and *REG*_FMK, *FIELD* must be a writable field.
   - ❑ For *REG*_FGET, the field must be a writable field.
3) *regval* indicates the value to write in the register (*REG*).
4) *fieldval* indicates the value to write in the field (*FIELD*).

Chapter 5

CHIP Module

The CSL chip module offers general CPU functions for C55x register accesses. The CHIP module is not handle–based.

Topic **Page**

5.1 Overview **5-2**
5.2 Function **5-3**

5.1 Overview

The CSL CHIP module offers general CPU functions. The CHIP module is not handle-based.

Table 5–1 lists the functions available as part of the CHIP module.

Table 5–1. CHIP Functions

Function	**Purpose**	**See page ...**
CHIP_getRevID	Returns the value of the RevID register.	5-3
CHIP_getEndian	Returns the endian mode of the device.	5-3
CHIP_getDieID_High32	Returns the high 32 bits of the DieID register.	5-3
CHIP_getDieID_Low32	Returns the low 32 bits of the DieID register.	5-4

5.2 Functions

This section lists the functions in the CHIP module.

CHIP_getRevID

Get Rev ID

Function Uint16 CHIP_getRevID();

Arguments None

Return Value Rev ID

Description This function returns the Rev ID field of the CSR register.

Example

```
Uint16 RevId;
CpuId = CHIP_getCpuId();
...
RevId = CHIP_getRevID();
```

CHIP_getEndian

Get endian mode (C5416 and C5421 only)

Function Uint16 CHIP_getEndian();

Arguments None

Return Value endian mod CHIP_ENDIAN_LITTLE = 1

Description Returns the current endian mode of the device as determined by the EN bit of the CSR register.

Example

```
UINT16 Endian;
…
Endian = CHIP_getEndian();
```

CHIP_getDieID_High32

Get the high 32 bits of the DieID register

Function Uint32 CHIP_getDieID_High32

Arguments None

Return Value high 32 bits of DieID

Description Returns high 32 bits of the DieID register

Example

```
Uint32 DieID_32_High;
…
DieID_32_high = CHIP_getDieID_High32();
```

CHIP_getDieID_Low32

Get the low 32 bits of the DieID register

Function Uint32 CHIP_getDieID_Low32

Arguments None

Return Value low 32 bits of DieID

Description Returns low 32 bits of the DieID register

Example

```
Uint32 DieID_32_High;
...
DieID_32_high = CHIP_getDieID_High32();
```

CHIP_getSubsysId

Get subsystem ID (5440 only)

Function Uint32 CHIP_SubsysId();

Arguments None

Return Value Subsytem ID

Description Get the sub-system ID (or core) from the a multi-core device

Example

```
Uint32 RevId;
RevId = CHIP_getRevId();
```

Chapter 6

DAT Module

The handle-based DAT (data) module allows you to use DMA Hardware to move data.

Topic **Page**

6.1 Overview 6-2
6.2 Functions 6-3

6.1 Overview

The handle-based DAT (data) module allows you to use DMA hardware to move data. This module works the same for all devices that support DMA regardless of the type of DMA controller. Therefore, any application code using the DAT module is compatible across all devices as long as the DMA supports the specific address reach and memory space.

The DAT copy operations occur on dedicated DMA hardware independent of the CPU. Because of this asynchronous nature, you can submit an operation to be performed in the background while the CPU performs other tasks in the foreground. Then you can use the DAT_wait() function to block completion of the operation before moving to the next task.

Since the DAT module uses the DMA peripheral, it cannot use a DMA channel that is already allocated by the application. To ensure this does not happen, you must call the DAT_open() function to allocate a DMA channel for exclusive use. When the module is no longer needed, you can free the DMA resource by calling DAT_close().

Table 6–1 lists the functions for use with the DAT modules. The functions are presented in the order that they will typically be used in an application.

Table 6–1 summarizes the primary DAT functions.

- ❑ Your application must call DAT_open() and DAT_close().
- ❑ Your application can also call DAT_copy(), DAT_copy2D(), DAT_fill(), DAT_wait().

Table 6–1. DAT Primary Summary

(a) DAT Functions

Function	Purpose	See page ...
DAT_copy()	Copies data of specific length from the source memory to the destination memory.	6-3
DAT_copy2D()	Copies data of specific line length from the source memory to the destination memory.	6-3
DAT_fill()	Fills the destination memory with a data value	6-4
DAT_wait()	DAT wait function	6-4
DAT_open()	Opens the DAT with a channel number and a channel priority	6-5
DAT_close()	Closes the DAT	6-5

6.2 Functions

This section describes the functions in the DAT module.

DAT_copy

Performs bytewise copy from source to destination memory

Function

Uint16 DAT_copy(DAT_Handle hDat,
    (DMA_AdrPtr)Src
    (DMA_AdrPtr)Dst
    Uint16 ElemCnt
);

Arguments

Src	Pointer to source memory assumes byte addresses
Dst	Pointer to destination memory assumes byte addresses
ByteCnt	Number of bytes to transfer to *Dst

Return Value

Uint 16 ID   Transfer ID

Description

Copies the memory values from the Src to the Dst memory locations.

Example

```
DAT_copy(hset
   (DMA_AdrPtr)0xF000, /* src     */
   (DMA_AdrPtr)0xFF00, /* dst     */
   0x0010              /* ByteCnt */
)
```

DAT_copy2D

Copies 2–dimensional data from source memory to destination memory

Function

Uint16 DAT_copy2D(DAT_Handle hDat,
    Uint16 Type
    (DMA_AdrPtr)Src
    (DMA_AdrPtr)Dst
    Uint16 Line
    Uint16 LineCnt
    Uint16 LinePitch
);

Arguments

Type	Type of Transfer: DAT_1D2D, DAT_2D1D, DAT2D2D
Src	Pointer to source memory assumes byte addresses
Dst	Pointer to destination memory assumes byte addresses
Line	Number of 16-bit words in one line
LineCnt	Number of lines to copy
LinePitch	Holds an index of what lines to copy next

Return Value

Uint 16 ID   Transfer ID

Description

Copies the memory values from the Src to the Dst memory locations.

Example

```
DAT_copy2D(hDat,
   DAT2D2D,            /* Type */
   (DMA_AdrPtr)0xFF00, /* src      */
   (DMA_AdrPtr)0xF000, /* dst      */
   0x0010,             /* linelen  */
   0x0004,             /* Line Cnt */
   0x0101,             /*LinePitch */
)
```

DAT_fill

Fills DAT destination memory with value

Function

void DAT_fill(DAT_Handle hDat,
(DMA_AdrPtr)Dst
Uint16 ElemCnt
Uint16 *Value
);

Arguments

*Dst	Pointer to destination memory location
Cnt	Number of 16-bit words to fill
*Value	Pointer to value that will fill the memory

Return Value

Uint 32 ID Transfer ID

Description

Fills the destination memory with a value for a specified byte count

Example

```
Dat_fill(hDat,
(DMA_AdrPtr)0x00FF,    /* dst     */
0x0010,                /* ElemCnt */
0xFFFF,                /*Value    */
);
```

DAT_wait

DAT wait function

Function

void DAT_reset
DAT_Handle hDat,
);

Arguments

hDat Device handler (see DAT_open).

Return Value

none

Description

Recursive functions that waits

Example

```
Dat_wait(my hDat);
```

DAT_open — *Opens DAT for DAT calls*

Function

```
DAT_Handle DAT_open(
    int ChaNum
    int  Priority
    Uint32 flags
);
```

Arguments

ChaNum	Channel Number: DAT_CHA0, DAT_CHA1, DAT_CHA2, DAT_CHA3, DAT_CHA4, DAT_CHA5,
Priority	Channel Priority Number: DAT_PRI_LOW DAT_PRI_HIGH
Flags	Event Flag

Return Value None

Description Before a DAT channel can be used, it must first be opened by this function with an assigned priority. Once opened, it cannot be opened again until closed (see DAT_close).

Example

```
DAT_open(DAT_CHA0,DAT_PRI_LOW,0);
```

DAT_close — *Closes DAT*

Function

```
void DAT_close(DAT_Handle hDat
);
```

Arguments hDat

Return Value None

Description Closes a previously opened DAT device. DAT event is disabled and cleared.

Example

```
Dat_close(hDat);
```

Chapter 7

DMA Module

This chapter describes the structure, functions, and macros of the DMA module.

Topic **Page**

7.1	Overview	7-2
7.2	Configuration Structure	7-4
7.3	Functions	7-6
7.4	Macros	7-11
7.5	Examples	7-13

7.1 Overview

Table 7–1 summarizes the primary API functions.

- Your application must call DMA_open() and DMA_close().
- Your application can also call DMA_reset(hDma).
- You can perform configuration by calling either DMA_config(), DMA_configArgs(), or any of the SET register macros.

  Because DMA_config() and DMA_configArgs() initialize 11 control registers, macros are provided to enable efficient access to individual registers when you need to set only one or two.

  Using DMA_config() to initialize the DMA registers is the recommended approach.

The DMA API defines macros designed for the following primary purposes:

- The *RMK* macros create individual control-register masks for the following purposes:
  - To initialize an DMA_Config structure that you then pass to functions such as DMA_config().
  - To use as arguments for functions such as DMA_configArgs()
  - To use as arguments for the appropriate SET macro.
- Other macros are available primarily to facilitate reading and writing individual bits and fields in the DMA control registers.

Table 7–1 (c) lists the most commonly used macros. Section 7.4 includes a description of all DMA macros.

Table 7–1. DMA Primary Summary

(a) DMA Configuration Structure

Structure	Purpose	See page...
DMA_Config	DMA configuration structure used to setup the DMA interface	7-4

(b) DMA Functions

Function	Purpose	See page ...
DMA_config()	Sets up DMA using configuration structure (DMA_Config)	7-6
DMA_configArgs()	Sets up DMA using register values passed to the function	7-7
DMA_getEventId()	Returns the IRQ Event ID for the DMA completion interrupt	7-9
DMA_open()	Opens the DMA and assigns a handler to it	7-9
DMA_close()	Closes the DMA and its corresponding handler	7-10
DMA_reset()	Resets the DMA registers with default values	7-10

(c) DMA Macros

Macro	Purpose	See page ...
DMA_XXXX_RMK	Creates a value to store in a particular register (See below)	
DMA_RSET(XXXX,Val)	Write a value to store in a particular register (See below)	
DMA_RGET(XXXX)	Read a value from a particular register (See below)	
	where XXXX is GCR,CSDP, CCR, CICR, CSR, CSSA_L, CSSA_U, CDSA_L, CDSA_U, CEN, CFN, CFI, CEI	

7.2 Configuration Structure

This section describes the structure in the DMA module.

DMA_Config — *DMA configuration structure used to set up DMA interface*

Structure DMA_Config

Members

Uint16 dmacsdp	DMA Channel Control Register
Uint16 dmaccr	DMA Channel Interrupt Register
Uint16 dmacicr	DMA Channel Status Register
(DMA_AdrPtr) dmacssal	DMA Channel Source Start Address (Lower Bits)
Uint16 dmacssau	DMA Channel Source Start Address (Upper Bits)
(DMA_AdrPtr) dmacdsal	DMA Channel Source Destination Address (Lower Bits)
Uint16 dmacdsau	DMA Channel Source Destination Address (Upper Bits)
Uint16 dmacen	DMA Channel Element Number Register
Uint16 dmacfn	DMA Channel Frame Number Register
Uint16 dmacfi	DMA Channel Frame Index Register **(CHIP_5510PG1_0)**
Uint16 dmacei	DMA Channel Element Index Register **(CHIP_5510PG1_0)**
Uint16 dmacsfi	DMA Channel Source Frame Index Register **(CHIP_5510PG2_0)**
Uint16 dmacsei	DMA Channel Source Element Index Register **(CHIP_5510PG2_0)**
Uint16 dmacdfi	DMA Channel Destination Frame Index Register **(CHIP_5510PG2_0)**
Uint16 dmacdei	DMA Channel Destination Element Index Register **(CHIP_5510PG2_0)**

Note: If the user selects –dCHIP_5510PG1_0, old registers will be given. If –dCHIP_5510PG2_0 is selected, then a new set of registers (as shown above) will be given.

Description DMA configuration structure used to set up the DMA Interface. You create and initialize this structure and then pass its address to the DMA_configArgs() function. You can use literal values or the *DMA_RMK* macros to create the structure member values.

Example

```
DMA_Config Config1 = {
0xFFFF,                /*dmacsdp*/
0xFFFF,                /*dmaccr*/
0xFFFF,                /*dmacicr*/
(DMA_AdrPtr)0xFFFF,    /*DMA_AdrPtr*/
0xFFFF,                /*dmacssau*/
(DMA_AdrPtr)0xFFFF,    /*DMA_AdrPtr*/
0xFFFF,                /*dmacdsau*/
0xFFFF,                /*dmacen*/
0xFFFF,                /*dmacfn*/
0xFFFF,                /*dmacfi*/
0xFFFF,                /*dmacei*/
}
```

7.3 Functions

This section describes the functions in the DMA module.

DMA_config — *Writes value to up DMA using configuration structure*

Function

```
void DMA_config(DMA_Handle hDma;
    DMA_Config *Config
);
```

Arguments

hDma	DMA Device handle
Config	Pointer to an initialized configuration structure

Return Value

None

Description

Writes a value to the DMA using the configuration structure. The values of the structure are written to the port registers. See also DMA_configArgs() and DMA_Config.

Example

```
DMA_Config MyConfig = {
0xFFFF,               /*dmacsdp*/
0xFFFF,               /*dmaccr*/
0xFFFF,               /*dmacicr*/
0xFFFF,               /*dmacssal*/
(DMA_AdrPtr) 0xFFFF,   /*dmacssau*/
0xFFFF,               /*dmacdsal*/
(DMA_AdrPtr) 0xFFFF,   /*dmacdsau*/
0xFFFF,               /*dmacen*/
0xFFFF,               /*dmacfn*/
0xFFFF,               /*dmacfi*/
0xFFFF,               /*dmacei*/
}
   DMA_config(hDma,&MyConfig);
```

DMA_configArgs *Writes to DMA using register values passed to function*

Function

Uint16 dmacsdp DMA Channel Control Register
Uint16 dmaccr DMA Channel Interrupt Register
Uint16 dmacicr DMA Channel Status Register
Uint16 dmacssal DMA Channel Source Start Address (Lower Bits)
Uint16 dmacssau DMA Channel Source Start Address (Upper Bits)
Uint16 dmacdsal DMA Channel Source Destination Address (Lower Bits)
Uint16 dmacdsau DMA Channel Source Destination Address (Upper Bits)
Uint16 dmacen DMA Channel Element Number Register
Uint16 dmacfn DMA Channel Frame Number Register
Uint16 dmacfi DMA Channel Frame Index Register **(CHIP_5510PG1_0)**
Uint16 dmacei DMA Channel Element Index Register **(CHIP_5510PG1_0)**
Uint16 dmacsfi DMA Channel Source Frame Index Register **(CHIP_5510PG2_0)**
Uint16 dmacsei DMA Channel Source Element Index Register **(CHIP_5510PG2_0)**
Uint16 dmacdfi DMA Channel Destination Frame Index Register **(CHIP_5510PG2_0)**
Uint16 dmacdei DMA Channel Destination Element Index Register **(CHIP_5510PG2_0)**

Arguments

For **CHIP_5510PG1_0**:

```
void DMA_configArgs(DMA_Handle hDma
      0xFFFF,    /*dmacsdp*/
      0xFFFF,    /*dmaccr*/
      0xFFFF,    /*dmacicr*/
      0xFFFF,    /*dmacssal*/
      0xFFFF,    /*dmacssau*/
      0xFFFF,    /*dmacdsal*/
      0xFFFF,    /*dmacdsau*/
      0xFFFF,    /*dmacen*/
      0xFFFF,    /*dmacfn*/
      0xFFFF,    /*dmacfi*/
      0xFFFF,    /*dmacei*/
);
```

For **CHIP_5510PG2_0**:

```
void DMA_configArgs(DMA_Handle hDma
      0xFFFF,    /*dmacsdp*/
      0xFFFF,    /*dmaccr*/
```

```
        0xFFFF,    /*dmacicr*/
        0xFFFF,    /*dmacssal*/
        0xFFFF,    /*dmacssau*/
        0xFFFF,    /*dmacdsal*/
        0xFFFF,    /*dmacdsau*/
        0xFFFF,    /*dmacsfi*/
        0xFFFF,    /*dmacsei*/
        0xFFFF,    /*dmacdfi*/
        0xFFFF,    /*dmacdei*/
);
```

Return Value None

Description Writes to the DMA using the register values passed to the function. The register values are written to the DMA registers.

You may use literal values for the arguments; or for readability, you may use the DMA_RMK macros to create the register values based on field values.

Example

```
DMA_configArgs (
0xFFFF,     /*dmacsdp*/
0xFFFF,     /*dmaccr*/
0xFFFF,     /*dmacicr*/
0xFFFF,     /*dmacssal*/
0xFFFF,     /*dmacssau*/
0xFFFF,     /*dmacdsal*/
0xFFFF,     /*dmacdsau*/
0xFFFF,     /*dmacen*/
0xFFFF,     /*dmacfn*/
0xFFFF,     /*dmacfi*/
0xFFFF,     /*dmacei*/
);
```

DMA_getEventId — *Returns IRQ Event ID for DMA completion interrupt*

Function

Uint16 DMA_getEventId(
  DMA_Handle hDma
);

Arguments

hDma	Handle to DMA channel; see DMA_open().

Return Value

Event ID	IRQ Event ID for DMA Channel

Description

Returns the IRQ Event ID for the DMA completion interrupt. Use this ID to manage the event using the IRQ module.

Example

```
EventId = DMA_getEventId(hDma);
IRQ_enable(EventId);
```

For a complete example, see Section 7.5, Example 2.

DMA_open — *Opens DMA for DMA calls*

Function

DMA_Handle DMA_open(
  int ChaNum
  Uint32 flags
);

Arguments

ChaNum	DMA Channel Number: DMA_CHA0, DMA_CHA1 DMA_CHA2, DMA_CHA3, DMA_CHA4, DMA_CHA5
flags	Event Flag Number: Logical open or DMA_OPEN_RESET

Return Value

DMA_Handle	Device handler

Description

Before a DMA device can be used, it must first be opened by this function. Once opened, it cannot be opened again until closed (see DMA_close). The return value is a unique device handle that is used in subsequent DMA API calls. If the function fails, INV is returned. If the DMA_OPEN_RESET is specified, then the power on defaults are set and any interrupts are disabled and cleared.

Example

```
DMA_Handle hDma;
...
hDma = DMA_open(DMA_CHA0,0);
```

DMA_close — *Closes DMA*

Function

```
void DMA_close
    DMA_Handle hDma
 );
```

Arguments

hDma	Device Handle, see DMA_open();

Return Value

DMA_Handle	Device handler

Description

Closes a previously opened DMA device. The DMA event is disabled and cleared. The DMA registers are set to their default values.

Example

```
DMA_close(hDma);
```

DMA_reset — *Resets DMA*

Function

```
void DMA_reset
    DMA_Handle hDma
 );
```

Arguments

hDma	Device handle, see DMA_open();

Return Value

None

Description

Resets the DMA device. Disables and clears the interrupt event and sets the DMA registers to default values. If INV is specified, all DMA devices are reset.

Example

```
DMA_reset(hDma);
```

7.4 Macros

CSL offers a collection of macros to gain individual access to the DMA peripheral registers and fields.

Table 7–2 contains a list of macros available for the DMA module. To use them, include ”csl_dma.h”.

Table 7–2. DMA CSL Macros Using DMA Port Number

(a) Macros to read/write DMA register values

Macro	Syntax
DMA_RGET()	Uint16 DMA_RGET(*REG*)
DMA_RSET()	Void DMA_RSET(*REG*, Uint16 *regval*)

(b) Macros to read/write DMA register field values (Applicable only to registers with more than one field)

Macro	Syntax
DMA_FGET()	Uint16 DMA_FGET(*REG*, *FIELD*)
DMA_FSET()	Void DMA_FSET(*REG*, *FIELD*, Uint16 *fieldval*)

(c) Macros to create value to DMA registers and fields (Applies only to registers with more than one field)

Macro	Syntax
DMA_REG_RMK()	Uint16 DMA_REG_RMK(*fieldval_n*,…*fieldval_0*) Note: *Start with field values with most significant field positions: field_n: MSB field field_0: LSB field *only writable fields allowed
DMA_FMK()	Uint16 DMA_FMK(*REG*, *FIELD*, *fieldval*)

(d) Macros to read a register address

Macro	Syntax
DMA_ADDR()	Uint16 DMA_ADDR(*REG*)

Notes: 1) *REG* indicates the register, xxx xxx.

2) *FIELD* indicates the register field name as specified in Appendix A.
- ❏ For *REG*_FSET and *REG*_FMK, *FIELD* must be a writable field.
- ❏ For *REG*_FGET, the field must be a writable field.

3) *regval* indicates the value to write in the register (*REG*).

4) *fieldval* indicates the value to write in the field (*FIELD*).

Table 7–3. DMA CSL Macros Using Handle

(a) Macros to read/write DMA register values

Macro	Syntax
DMA_RGET_H()	Uint16 DMA_RGET_H(DMA_Handle hDMA, *REG*)
DMA_RSET_H()	Void DMA_RSET_H(DMA_Handle hDMA, *REG*, Uint16 *regval*)

(b) Macros to read/write DMA register field values (Applicable only to registers with more than one field)

Macro	Syntax
DMA_FGET_H()	Uint16 DMA_FGET_H(DMA_Handle hDMA, *REG*, *FIELD*)
DMA_FSET_H()	Void DMA_FSET_H(DMA_Handle hDMA, *REG*, *FIELD*, *Uint16 fieldval*)

(c) Macros to read a register address

Macro	Syntax
DMA_ADDR_H()	Uint16 DMA_ADDR_H(DMA_Handle hDMA, *REG*)

Notes:

1) *REG* indicates the register, xxx xxx.
2) *FIELD* indicates the register field name as specified in Appendix A.
   - For *REG*_FSET_H, *FIELD* must be a writable field.
   - For *REG*_FGET_H, the field must be a writable field.
3) *regval* indicates the value to write in the register (*REG*).
4) *fieldval* indicates the value to write in the field (*FIELD*).

7.5 Examples

The following CSL DMA initialization examples are provided under the \examples\dma1, dma2 directories.

- Example 1 DMA channel initialization using DMA_config()
- Example 2 DMA channel initialization using DMA_configArgs()

For illustration purposes, Example 1 is covered in detail below, and is illustrated in Figure 7–1, on page 7-13.

Example 1 explains how DMA Channel 0 is initialized to transfer a data table from 0x3000@data space to 0x2000@data space. Basic initialization values are as follows:

- Source address: 2000h in data space
- Destination address: 3000h in data space
- Transfer size: 10h words single words

The following three macros are used to create the initialization values for DMACSDP, DMACCR and DMACICR respectively:

DMA_DMMCR_RMK(autoinit, dinm, imod, ctmod, sind, dms, dind, dmd)
0 0 0 0 1 1 1 1

DMA_DMSFC_RMK(dsyn, dblw, framecount)
0 0 0 (single-frame, Nframes-1)

DMA_DMACSDP_RMK(dstben,dstpack,dst,srcben,srcpack,src,datatype)
0 0 0 0 0 0 1
DMA_DMACCR_RMK
(dstamode,srcamode,endprog,fifoflush,repeat,autoinit,en,prio,fs,sync)
1 1 0 0 0 0 0 0 0 0
DMA_DMACICR_RMK(blockie,lastie,frameie,firsthalfie,dropie,timeoutie)
1 1 1 1 1 1

Figure 7–1. DMA Channel Initialization Using DMA_config()

```
/*============================================================*\
* Include HDMA macros and symbol definitions
\*============================================================*/
#include <csl.h>
#include <csl_irq.h>
#include <csl_dma.h>
```

```
/*============================================================*\
* Define size constant
\*============================================================*/

#define N 128

/*============================================================*\
* Define Source Data Table to be located at
* 3000@ps. This will be assigned at link time
\*============================================================*/

#pragma DATA_SECTION(src,"table1")
Uint16 src[N];

/*============================================================*\
* Define Destination Data Table to be located at
* 2000@ds. This will be assigned at link time
\*============================================================*/

#pragma DATA_SECTION(dst, "table2")
Uint16 dst[N];


/*============================================================*\
\*============================================================*/

/*============================================================*/

/*============================================================*\
* The following three macros are used to create the initiali-
zation
* values for DMACSDP and DMACSDP respectively:
*
* DMA_DMACSDP_RMK(dstben,dstpack,dst,srcben,srcpack,src,data-
type)
*                  0        0     0   0        0     0      1
*
* DMA_DMACCR_RMK
(dstamode,srcamode,endprog,fifoflush,repeat,autoin-
it,en,prio,fs,sync)
*                  1        1        0        0       0
0      0   0   0  0
*
* DMA_DMACICR_RMK(blockie,lastie,frameie,firsthalfie,dro-
pie,timeoutie)
*                1       1       1        1        1        1
\*===========================================================*/

DMA_Config  myconfig = {
    DMA_DMACSDP_RMK(
      DMA_DMACSDP_DSTBEN_NOBURST,
      DMA_DMACSDP_DSTPACK_OFF,
      DMA_DMACSDP_DST_DARAM,
```

```
      DMA_DMACSDP_SRCBEN_NOBURST,
      DMA_DMACSDP_SRCPACK_OFF,
      DMA_DMACSDP_SRC_DARAM,
      DMA_DMACSDP_DATATYPE_16BIT),         /* DMACSDP  */

    DMA_DMACCR_RMK(
      DMA_DMACCR_DSTAMODE_POSTINC,
      DMA_DMACCR_SRCAMODE_POSTINC,
      DMA_DMACCR_ENDPROG_OFF,
      DMA_DMACCR_FIFOFLUSH_OFF,
      DMA_DMACCR_REPEAT_OFF,
      DMA_DMACCR_AUTOINIT_OFF,
      DMA_DMACCR_EN_STOP,
      DMA_DMACCR_PRIO_HI,
      DMA_DMACCR_FS_ENABLE,
      DMA_DMACCR_SYNC_None),               /* DMACCR   */

    DMA_DMACICR_RMK(
      DMA_DMACICR_BLOCKIE_OFF,
      DMA_DMACICR_LASTIE_OFF,
      DMA_DMACICR_FRAMEIE_ON,
      DMA_DMACICR_FIRSTHALFIE_OFF,
      DMA_DMACICR_DROPIE_OFF,
      DMA_DMACICR_TIMEOUTIE_OFF),          /* DMACICR  */

   (DMA_AdrPtr) &src,                      /* DMACSSAL */
    0,                                     /* DMACSSAU */
    (DMA_AdrPtr)&dst,                      /* DMACDSAL */
    0,                                     /* DMACDSAU */
    N,                                     /* DMACEN   */
    1,                                     /* DMACFN   */
    0,                                     /* DMACFI   */
    0};                                    /* DMACEI   */

/*============================================================*\
* Define a DMA_Handle pointer. DMA_open will return a pointer
to
* a DMA_Handle when a DMA channel is opened.
\*============================================================*/
DMA_Handle myhDma;
int i,j;
Uint16 err = 0;
volatile Bool WaitForTransfer = TRUE;

void main(void) {
  CSL_init();                                 /* Init CSL
    */

  for(i=0; i<= N–1; i++) {
   dst[i] = 0;
   src[i] = i+1;
  }
```

```
  myhDma = DMA_open(DMA_CHA0, 0);                 /* Open
Channel      */
  myconfig.dmacssal =
     (DMA_AdrPtr)(((Uint16)(myconfig.dmacssal))<<1);  /*
Change word addres to byte address */
  myconfig.dmacdsal =
     (DMA_AdrPtr)(((Uint16)(myconfig.dmacdsal))<<1);  /*
Change word address to byte address */
  DMA_config(myhDma, &myconfig);                  /* Configure
Channel */
  DMA_FSET_H(myhDma,DMACCR,EN,1);                          /*
Begin Transfer    */


    while(!DMA_FGET_H(myhDma,DMACSR,FRAME));   /* Wait for
complete */

 for (i = 0; i <= 10000; i++)
 { asm(" nop"); }

   for (i = 0; i <= N-1; i++) {
    if (dst[i] != src[i])
      ++err;
  }

  if (err)
    printf (">>> Warning, DMA Example 1 Failed. \n");
  else
    printf ("...DMA Example 1 Complete \n");


   DMA_close(myhDma);
}
```

Chapter 8

EMIF Module

This chapter describes the structure, functions, and macros of the EMIF module.

Topic **Page**

8.1 Overview 8-2

8.2 Configuration Structure 8-5

8.3 Functions 8-7

8.4 Macros 8-10

8.1 Overview

Table 8–1 summarizes the primary API functions.

You can perform configuration by calling either EMIF_config(), EMIF_configArgs(), or any of the SET register macros.

Because EMIF_config() and EMIF_configArgs() initialize 17 control registers, macros are provided to enable efficient access to individual registers when you need to set only one or two.

Using EMIF_config() to initialize the EMIF registers is the recommended approach.

The EMIF API defines macros designed for the following primary purposes:

The *RMK* macros create individual control-register masks for the following purposes:

❑ To initialize an EMIF_Config structure that is passed to EMIF_config().

❑ To use as arguments for functions such as EMIF_configArgs().

❑ To use as arguments for the appropriate SET macro.

❑ Other macros are available primarily to facilitate reading and writing individual bits and fields in the control registers.

Table 8–1 (c) lists the most commonly used macros. Section 8.4 includes a description of all EMIF macros.

Table 8–1. EMIF Primary Summary

(a) EMIF Configuration Structure

Structure	Purpose	See page...
EMIF_Config	EMIF configuration structure used to setup the EMIF interface	8-5

(b) EMIF Functions

Function	Purpose	See page ...
EMIF_config()	Sets up EMIF using configuration structure (EMIF_Config)	8-7
EMIF_configArgs()	Sets up EMIF using register values passed to the function	8-8

(c) EMIF Macros

Macro	Purpose	See page ...
EMIF_XXXX_RMK	Creates a value to store in a particulat register (See below)	
EMIF_RSET(XXXX, Val)	Write a value to store in a particular register (See below)	
EMIF_RGET(XXXX)	Read a value from a particular register (See Below)	
	where XXXX is GCR, GRR, CEO1, CEO2, CEO3, CE11, CE12, CE13, CE21, CE22, CE23, CE31, CE32, CE33, SDRAMCR1, SDRAMPR, SDRAMIR, SDRAMCR2	

8.2 Configuration Structure

This section describes the structure in the EMIF module.

EMIF_Config — *EMIF configuration structure used to set up EMIF interface*

Structure EMIF_Config

Members

Uint16 egcr	Global Control Register
Uint16 egrst	Global Reset Register
Uint16 ce01	EMIF CE0 Space Control Register 1
Uint16 ce02	EMIF CE0 Space Control Register 2
Uint16 ce03	EMIF CE0 Space Control Register 3
Uint16 ce11	EMIF CE1 Space Control Register 1
Uint16 ce12	EMIF CE1 Space Control Register 2
Uint16 ce13	EMIF CE1 Space Control Register 3
Uint16 ce21	EMIF CE2 Space Control Register 1
Uint16 ce22	EMIIF CE2 Space Control Register 2
Uint16 ce23	EMIF CE2 Space Control Register 3
Uint16 ce31	EMIF CE3 Space Control Register 1
Uint16 ce32	EMIF CE3 Space Control Register 2
Uint16 ce33	EMIF CE3 Space Control Register 3
Uint16 sdc1	EMIF SDRAM Control Register 1
Uint16 sdprd	EMIF SDRAM Period Register
Uint16 sdinit	EMIF SDRAM Initialization Register
Uint16 sdc2	EMIF SDRAM Control Register 2

Description The EMIF configuration structure is used to set up the EMIF Interface. You create and initialize this structure and then pass its address to the EMIF_config() function. You can use literal values or the *EMIF_RMK* macros to create the structure member values.

Example

```
EMIF_Config Config1 = {
   0x06CF, /* egcr */
   0xFFFF, /* egrst */
   0x7FFF, /* ce01 */
   0xFFFF, /* ce02 */
   0x00FF, /* ce03 */
   0x7FFF, /* ce11 */
   0xFFFF, /* ce12 */
   0x00FF, /* ce13 */
   0x7FFF, /* ce21 */
   0xFFFF, /* ce22 */
   0x00FF, /* ce23 */
   0x7FFF, /* ce31 */
   0xFFFF, /* ce32 */
   0x00FF, /* ce33 */
   0x07FF, /* sdc1 */
   0x0FFF, /* sdprd */
   0x07FF, /* sdinit */
   0x03FF, /* sdc2 */
}
```

8.3 Functions

This section describes the functions in the EMIF module.

EMIF_config — *Writes value to up EMIF using configuration structure*

Function

```
void EMIF_config(
    EMIF_Config *Config
);
```

Arguments

Config   Pointer to an initialized configuration structure

Return Value

None

Description

Writes a value to up the EMIF using the configuration structure. The values of the structure are written to the port registers (see also EMIF_configArgs() and EMIF_Config).

Example

```
EMIF_Config MyConfig = {
   0x06CF, /* egcr */
   0xFFFF, /* egrst */
   0x7FFF, /* ce01 */
   0xFFFF, /* ce02 */
   0x00FF, /* ce03 */
   0x7FFF, /* ce11 */
   0xFFFF, /*  ce12  */
   0x00FF, /* ce13 */
   0x7FFF, /* ce21 */
   0xFFFF, /* ce22 */
   0x00FF, /* ce23 */
   0x7FFF, /* ce31 */
   0xFFFF, /* ce32 */
   0x00FF, /* ce33 */
   0x07FF, /* sdc1 */
   0x0FFF, /* sdprd */
   0x07FF, /* sdinit */
   0x03FF, /* sdc2 */ }

EMIF_config(&MyConfig);
```

EMIF_configArgs Writes to EMIF using register values passed to function

Function

```
void EMIF_configArgs(
Uint16 egcr,
Uint16 egrst,
Uint16 ce01,
Uint16 ce02,
Uint16 ce03,
Uint16 ce11,
Uint16 ce12,
Uint16 ce13,
Uint16 ce21,
Uint16 ce22,
Uint16 ce23
Uint16 ce31,
Uint16 ce32,
Uint16 ce33,
Uint16 sdc1,
Uint16 sdprd,
Uint16 sdinit,
Uint16 sdc2,)
```

Arguments

egcr	Global Control Register
egrst	Global Reset Register
ce01	EMIF CE0 Space Control Register 1
ce02	EMIF CE0 Space Control Register 2
ce03	EMIF CE0 Space Control Register 3
ce11	EMIF CE1 Space Control Register 1
ce12	EMIF CE1 Space Control Register 2
ce13	EMIF CE1 Space Control Register 3
ce21	EMIF CE2 Space Control Register 1
ce22	EMIIF CE2 Space Control Register 2
ce23	EMIF CE2 Space Control Register 3
ce31	EMIF CE3 Space Control Register 1
ce32	EMIF CE3 Space Control Register 2
ce33	EMIF CE3 Space Control Register 3
sdc1	EMIF SDRAM Control Register 1
sdprd	EMIF SDRAM Period Register
sdinit	EMIF SDRAM Initialization Register
sdc2	EMIF SDRAM Control Register 2

Return Value None

Description Writes to the EMIF using the register values passed to the function. The register values are written to the EMIF registers.

You may use literal values for the arguments; or for readability, you may use the EMIF_RMK macros to create the register values based on field values.

Example

```
EMIF_configArgs (
   0x06CF, /* egcr */
   0xFFFF, /* egrst */
   0x7FFF, /* ce01 */
   0xFFFF, /* ce02 */
   0x00FF, /* ce03 */
   0x7FFF, /* ce11 */
   0xFFFF, /* ce12 */
   0x00FF, /* ce13 */
   0x7FFF, /* ce21 */
   0xFFFF, /* ce22 */
   0x00FF, /* ce23 */
   0x7FFF, /* ce31 */
   0xFFFF, /* ce32 */
   0x00FF, /* ce33 */
   0x07FF, /* sdc1 */
   0x0FFF, /* sdprd */
   0x07FF, /* sdinit */
   0x03FF, /* sdc2 */
)
```

8.4 Macros

CSL offers a collection of macros to gain individual access to the EMIF peripheral registers and fields.

Table 8–2 contains a list of macros available for the EMIF module. To use them, include ”csl_emif.h”.

Table 8–2. EMIF CSL Macros using EMIF Port Number

(a) Macros to read/write EMIF register values

Macro	Syntax
EMIF_RGET()	Uint16 EMIF_RGET(*REG*)
EMIF_RSET()	Void EMIF_RSET(*REG*, Uint16 *regval*)

(b) Macros to read/write EMIF register field values (Applicable only to registers with more than one field)

Macro	Syntax
EMIF_FGET()	Uint16 EMIF_FGET(*REG*, *FIELD*)
EMIF_FSET()	Void EMIF_FSET(*REG*, *FIELD*, Uint16 *fieldval*)

(c) Macros to create value to EMIF registers and fields (Applies only to registers with more than one field)

Macro	Syntax
EMIF_REG_RMK()	Uint16 EMIF_REG_RMK(*fieldval_n*,…*fieldval_0*) Note: *Start with field values with most significant field positions: field_n: MSB field field_0: LSB field *only writable fields allowed
EMIF_FMK()	Uint16 EMIF_FMK(*REG*, *FIELD*, *fieldval*)

(d) Macros to read a register address

Macro	Syntax
EMIF_ADDR()	Uint16 EMIF_ADDR(*REG*)

Notes:
1) *REG* indicates the register, xxx xxx.
2) *FIELD* indicates the register field name as specified in Appendix A.
   - ❑ For *REG*_FSET and *REG*_FMK, *FIELD* must be a writable field.
   - ❑ For *REG*_FGET, the field must be a writable field.
3) *regval* indicates the value to write in the register (*REG*).
4) *fieldval* indicates the value to write in the field (*FIELD*).

Chapter 9

GPIO Module

The GPIO module is designed to allow central control of the non-multiplexed GPIO pins available in the C55x devices.

Topic **Page**

9.1 Overview 9-2

9.2 Macros 9-3

9.1 Overview

The GPIO module is designed to allow central control of the non-multiplexed GPIO pins available in the C55x devices.

Currently, there are no functions available in the GPIO module. Macros that allow register access have been provided.

9.2 Macros

CSL offers a collection of macros to gain individual access to the GPIO peripheral registers and fields.

Table 9–1 contains a list of macros available for the GPIO module. To use them, include ”csl_gpio.h”.

Table 9–1. GPIO CSL Macros Using GPIO Port Number

(a) Macros to read/write GPIO register values

Macro	Syntax
GPIO_RGET()	Uint16 GPIO_RGET(*REG*)
GPIO_RSET()	Void GPIO_RSET(*REG*, Uint16 *regval*)

(b) Macros to read/write GPIO register field values (Applicable only to registers with more than one field)

Macro	Syntax
GPIO_FGET()	Uint16 GPIO_FGET(*REG*, *FIELD*)
GPIO_FSET()	Void GPIO_FSET(*REG*, *FIELD*, Uint16 *fieldval*)

(c) Macros to create value to GPIO registers and fields (Applies only to registers with more than one field)

Macro	Syntax
GPIO_REG_RMK()	Uint16 GPIO_REG_RMK(*fieldval_n*,…*fieldval_0*) Note: *Start with field values with most significant field positions: field_n: MSB field field_0: LSB field *only writable fields allowed
GPIO_FMK()	Uint16 GPIO_FMK(*REG*, *FIELD*, *fieldval*)

(d) Macros to read a register address

Macro	Syntax
GPIO_ADDR()	Uint16 GPIO_ADDR(*REG*)

Notes:
1) *REG* include the registers IODIR and IODATA.
2) *FIELD* indicates the register field name as specified in Appendix A.
   - ❑ For *REG*_FSET and *REG*_FMK, *FIELD* must be a writable field.
   - ❑ For *REG*_FGET, the field must be a writeable field.
3) *regval* indicates the value to write in the register (*REG*).
4) *fieldval* indicates the value to write in the field (*FIELD*).

Chapter 10

IRQ Module

The IRQ module provides an easy to use interface for enabling/disabling interrupts.

Topic **Page**

10.1 Overview **10-2**

10.2 Configuration Structure **10-7**

10.3 Functions **10-8**

10.1 Overview

The IRQ module provides an interface for managing peripheral interrupts to the CPU. This API provides the following functionality:

- ❑ Masking an interrupt in the IMR register.
- ❑ Polling for the interrupt status from the IFR register.
- ❑ Placing the necessary code in the interrupt vector table to branch to a user-defined interrupt service routine (ISR).
- ❑ Enabling/Disabling Global Interrupts in the ST1 (INTM) bit.
- ❑ Reading and writing to parameters in the DSP/BIOS dispatch table. (When the DPS BIOS dispatcher option is enabled in DSP BIOS.)

  The DSP BIOS dispatcher is responsible for dynamically handling interrupts and maintains a table of ISRs to be executed for specific interrupts. The IRQ module has a set of APIs that update the dispatch table.

Table 10–2(a) and (b) list the primary and auxiliary IRQ functions. Table 10–2(c) lists the API functions that enable DSP/BIOS dispatcher communication. These functions should be used only when DSP/BIOS is present **and** the DSP/BIOS dispatcher is enabled. Table 10–3 lists available interrupts for this feature.

The IRQ functions in Table 10–2(a) can be used with or without DSP/BIOS; however, if DSP/BIOS is present, do not disable interrupts for long periods of time because this could disrupt the DSP/BIOS environment.

Table 10–2(b) lists the only API function that cannot be used when DSP/BIOS dispatcher is present or DSP/BIOS HWI module is used to configure the interrupt vectors . This function, IRQ_plug(), dynamically places code at the interrupt vector location to branch to a user–defined ISR for a specified event. If you call IRQ_plug() when DSP/BIOS dispatcher is present or HWI module has been used to configure interrupt vectors, this could disrupt the DSP/BIOS operating environment.

Table 10–1. IRQ Configuration Structure

Structure	Purpose	See page ...
IRQ_Config	IRQ structure that contains all local registers required to set up a specific IRQ channel.	10-7

Table 10–2. IRQ Functions

(a) Primary Functions

Function	Purpose	See page ...
IRQ_clear()	Clears the interrupt flag in the IFR register for the specified event.	10-8
IRQ_disable()	Disables the specified event in the IMR register.	10-9
IRQ_enable()	Enables the specified event in the IMR register flag.	10-10
IRQ_globalDisable()	Globally disables all maskable interrupts. (INTM = 1)	10-11
IRQ_globalEnable()	Globally enables all maskable interrupts. (INTM = 0)	10-11
IRQ_globalRestore()	Restores the status of global interrupt enable/disable (INTM).	10-12
IRQ_setVecs()	Sets the base address of the interrupt vector table.	10-14
IRQ_test()	Polls the interrupt flag in IFR register the specified event.	10-14

(b) Auxiliary Functions

IRQ_plug()	Writes the necessary code in the interrupt vector location to branch to the interrupt service routine for the specified event. **Caution:** Do not use this function when DSP/BIOS is present and the dispatcher is enabled.	10-7

(c) DSP/BIOS Dispatcher Communication Functions

IRQ_config()	Updates the DSP/BIOS dispatch table with a new configuration for the specified event.	10-8
IRQ_configArgs()	Updates the DSP/BIOS dispatch table with a new configuration for the specified event.	10-9
IRQ_getConfig()	Returns current DSP/BIOS dispatch table entries for the specified event.	10-10
IRQ_getArg()	Returns value of the argument to the interrupt service routine that the DSP/BIOS dispatcher passes when the interrupt occurs.	10-10
IRQ_map()	Maps a logical event to its physical interrupt.	10-12
IRQ_setArg()	Sets the value of the argument for DSP/BIOS dispatch to pass to the interrupt service routine for the specified event.	10-13

Table 10–3. IRQ_EVT_NNNN Event List

(a) IRQ Events

Constant	Purpose
IRQ_EVT_RS	Reset
IRQ_EVT_SINTR	Software Interrupt
IRQ_EVT_NMI	Non-Maskable Interrupt (NMI)
IRQ_EVT_SINT16	Software Interrupt #16
IRQ_EVT_SINT17	Software Interrupt #17
IRQ_EVT_SINT18	Software Interrupt #18
IRQ_EVT_SINT19	Software Interrupt #19
IRQ_EVT_SINT20	Software Interrupt #20
IRQ_EVT_SINT21	Software Interrupt #21
IRQ_EVT_SINT22	Software Interrupt #22
IRQ_EVT_SINT23	Software Interrupt #23
IRQ_EVT_SINT24	Software Interrupt #24
IRQ_EVT_SINT25	Software Interrupt #25
IRQ_EVT_SINT26	Software Interrupt #26
IRQ_EVT_SINT27	Software Interrupt #27
IRQ_EVT_SINT28	Software Interrupt #28
IRQ_EVT_SINT29	Software Interrupt #29
IRQ_EVT_SINT30	Software Interrupt #30
IRQ_EVT_SINT0	Software Interrupt #0
IRQ_EVT_SINT1	Software Interrupt #1
IRQ_EVT_SINT2	Software Interrupt #2
IRQ_EVT_SINT3	Software Interrupt #3
IRQ_EVT_SINT4	Software Interrupt #4
IRQ_EVT_SINT5	Software Interrupt #5
IRQ_EVT_SINT6	Software Interrupt #6

(a) IRQ Events

Constant	Purpose
IRQ_EVT_SINT7	Software Interrupt #7
IRQ_EVT_SINT8	Software Interrupt #8
IRQ_EVT_SINT9	Software Interrupt #9
IRQ_EVT_SINT10	Software Interrupt #10
IRQ_EVT_SINT11	Software Interrupt #11
IRQ_EVT_SINT12	Software Interrupt #12
IRQ_EVT_SINT13	Software Interrupt #13
IRQ_EVT_INT0	External User Interrupt #0
IRQ_EVT_INT1	External User Interrupt #1
IRQ_EVT_INT2	External User Interrupt #2
IRQ_EVT_INT3	External User Interrupt #3
IRQ_EVT_TINT0	Timer 0 Interrupt
IRQ_EVT_HINT	Host Interrupt (HPI)
IRQ_EVT_DMA0	DMA Channel 0 Interrupt
IRQ_EVT_DMA1	DMA Channel 1 Interrupt
IRQ_EVT_DMA2	DMA Channel 2 Interrupt
IRQ_EVT_DMA3	DMA Channel 3 Interrupt
IRQ_EVT_DMA4	DMA Channel 4 Interrupt
IRQ_EVT_DMA5	DMA Channel 5 Interrupt
IRQ_EVT_RINT0	MCBSP Port #0 Receive Interrupt
IRQ_EVT_XINT0	MCBSP Port #0 Transmit Interrupt
IRQ_EVT_RINT2	MCBSP Port #2 Receive Interrupt

(a) IRQ Events

Constant	Purpose
IRQ_EVT_XINT2	MCBSP Port #2 Transmit Interrupt
IRQ_EVT_TINT1	Timer #1 Interrupt
IRQ_EVT_HPINT	Host Interrupt (HPI)
IRQ_EVT_RINT1	MCBSP Port #1 Receive Interrupt
IRQ_EVT_XINT1	MCBSP Port #1 Transmit Interrupt
IRQ_EVT_IPINT	FIFO Full Interrupt
IRQ_EVT_SINT14	Software Interrupt #14
IRQ_EVT_WDTINT	Watchdog Timer Interrupt

10.2 Configuration Structure

IRQ_Config *IRQ configuration structure*

Structure IRQ_Config

Members
IRQ_IsrPtr funcAddr; *Address of interrupt service routine*
Uint32 ierMask; *Interrupt to disable the existing ISR*
Uint32 funcArg; *Argument to pass to ISR when invoked*

Description This is the IRQ configuration structure used to update a DSP/BIOS table entry. You create and initialize this structure then pass its address to the IRQ_config( ) function.

Example

```
IRQ_Config MyConfig = {
  0x0000, /* funcAddr */
  0x0300, /* ierMask  */
  0x0000, /* funcArg  */
};
```

10.3 Functions

This sections describes the IRQ functions.

IRQ_clear

Clears event flag from IFR register

Function

```
void IRQ_clear(
    Uint16 EventId
);
```

Arguments

EventId	Event ID, see IRQ_EVT_NNNN (Table 10–3) for a complete list of events. Or, use the PER_get XXX EventId() function to get the EventID.

Return Value

None

Description

Clears the event flag from the IFR register

Example

```
IRQ_clear(IRQ_EVT_TINT0);
```

IRQ_config

Clears event flag from IFR register

Function

```
void IRQ_config(
    Uint16 EventId,
    IRQ_Config *Config
);
```

Arguments

EventID	Event ID, see IRQ_EVT_NNNN for a complete list of events.
Config	Pointer to an initialized configuration structure

Return Value

None

Description

Updates the entry in the DSPBIOS dispatch table for the specified event.

Example

IRQ_configArgs — *Updates entry in DSPBIOS dispatch table*

Function

```
void IRQ_configArgs(
    Uint16 EventId,
    IRQ_IsrPtr funcAddr,
    Uint32 funcArg,
    Uint16 ierMask
);
```

Arguments

EventId	Event ID, see IRQ_EVT_NNNN for a complete list of events.
funcAddr	Interrupt service routine address
funcArg	Argument to pass to interrupt service routine when it is invoked by DSPBIOS dispatcher
ierMask	Interrupts to disable while processing the ISR for this event (Mask for IER0, IER1)

Return Value None

Description Updates DSPBIOS dispatch table entry for the specified event.

You may use literal values for the arguments. For readability, you may use the *RMK macros* to create the register values based on field values.

Example

```
IRQ_configArgs(EventID, funcAddr, funcArg, ierMask);
```

IRQ_disable — *Disables specified event*

Function

```
void IRQ_disable(
    Uint16 EventId
);
```

Arguments

EventId	Event ID, see IRQ_EVT_NNNN (Table 10–3) for a complete list of events. Or, use the PER_get XXX EventId() function to get the EventID.

Return Value None

Description Disables the specified event, by modifying the IMR register.

Example

```
IRQ_disable(IRQ_EVT_TINT0);
```

IRQ_enable

Enables specified event

Function

void IRQ_enable(
    Uint16 EventId
);

Arguments

EventId   Event ID, see IRQ_EVT_NNNN (Table 10–3) for a complete list of events. Or, use the PER_get XXX EventId() function to get the EventID.

Return Value

None

Description

Enables the specified event.

Example

```
IRQ_enable(IRQ_EVT_TINT0);
```

IRQ_getArg

Gets value for specified event

Function

Uint32 IRQ_getArg(
    Uint16 EventId
);

Arguments

EventId   Event ID, see IRQ_EVT_NNNN (Table 10–3) for a complete list of events. Or, use the PER_get XXX EventId() function to get the EventID.

Return Value

Value of argument

Description

Returns value for specified event.

Example

```
IRQ_getArg(IRQ_EVT_TINT0);
```

IRQ_getConfig

Gets DSP/BIOS dispatch table entry

Function

void IRQ_getConfig(
    Uint16 EventId,
    IRQ_Config *Config
);

Arguments

EventId   Event ID, see IRQ_EVT_NNNN (Table 10–3) for a complete list of events. Or, use the PER_get XXX EventId() function to get the EventID.

Config   Pointer to configuration structure

Return Value

None

Description

Returns current values in DSP/BIOS dispatch table entry for the specified event.

Example

IRQ_globalDisable *Globally Disables Interrupts*

Function

int IRQ_globalDisable(
);

Arguments

None

Return Value

intm   Returns the old INTM value

Description

This function globally disables interrupts by setting the INTM of the ST1 register. The old value of INTM is returned. This is useful for temporarily disabling global interrupts, then enabling them again.

Example

```
Uint32 intm;
intm = IRQ_globalDisable();
...
IRQ_globalRestore (intm);
```

IRQ_globalEnable *Globally Enables Interrupts*

Function

int IRQ_globalEnable(
);

Arguments

None

Return Value

intm   Returns the old INTM value

Description

This function globally Enables interrupts by setting the INTM of the ST1 register. The old value of INTM is returned. This is useful for temporarily enabling global interrupts, then disabling them again.

Example

```
Uint32 intm;
intm = IRQ_globalEnable();
...
IRQ_globalRestore (intm);
```

IRQ_globalRestore *Restores The Global Interrupt Mask State*

Function

void IRQ_globalRestore(
    int intm
);

Arguments

intm   Value to restore the INTM value to (0 = enable, 1 = disable)

Return Value

None

Description

This function restores the INTM state to the value passed in by writing to the INTM bit of the ST1 register. This is useful for temporarily disabling/enabling global interrupts, then restoring them back to its previous state.

Example

```
int intm;
intm = IRQ_globalDisable();
...
IRQ_globalRestore (intm);
```

IRQ_map *Maps Event To Physical Interrupt Number*

Function

void IRQ_map(
    Uint16 EventId
);

Arguments

EventId   Event ID, see IRQ_EVT_NNNN for a complete list of events.

Return Value

None

Description

This function maps a logical event to a physical interrupt number for use by DSPBIOS dispatch.

Example

```
IRQ_map(IRQ_EVT_TINT0);
```

IRQ_plug — *Initializes An Interrupt Vector Table Vector*

Function

int IRQ_plug(
    Uint16 EventId,
    IRQ_IsrPtr funcAddr,
);

Arguments

EventId	Event ID, see IRQ_EVT_NNNN (Table 10–3) for a complete list of events. Or, use the PER_get XXX EventId() function to get the EventID.
funcAddr	Address of the interrupt service routine to be called when the interrupt happens. This function must be C-collable and if implemented in C, it must be declared using the *interrupt* keyword.

Return Value

0 or 1

Description

Initializes an interrupt vector table vector with the necessary code to branch to the specified ISR.

Caution: Do not use this function when DSP/BIOS is present and the dispatcher is enabled.

Example

```
void  MyIsr ();
.
.
.
IRQ_plug (IRQ_EVT_TINT0, &myIsr)
```

IRQ_setArg — *Sets value of argument for DSPBIOS dispatch entry*

Function

void IRQ_setArg(
    Uint16   EventId
    Uint32   val
);

Arguments

EventId	Event ID, see IRQ_EVT_NNNN (Table 10–3) for a complete list of events. Or, use the PER_get XXX EventId() function to get the EventID.

Return Value

None

Description

Sets the argument that DSP/BIOS dispatcher will pass to the interrupt service routine for the specified event.

Example

```
IRQ_setArg(IRQ_EVT_TINT0, val);
```

IRQ_setVecs

Sets the base address of the interrupt vectors

Function

void IRQ_setVecs(
    Uint32 IVPD
    Uint32   val Uint32 IVPH
);

Arguments

vecs   IVPD pointer to the DSP interrupt vector table

Return Value

oldVecs   Returns IVPH Pointer to the Host interrupt Vector table

Description

Use this function to set the base address of the interrupt vector table in the IVPD and IVPH registers.

Caution: Changing the interrupt vector table base can have adverse effects on your system because you will be effectively eliminating all previous interrupt settings. There is a strong chance that the DSP/BIOS kernel and RTDX will fail if this function is not used with care.

Example

```
IRQ_setVecs ((void*) 0x8000);
```

IRQ_test

Tests event to see if its flag is set in IFR register

Function

Bool IRQ_test(
    Uint16 EventId
);

Arguments

EventId   Event ID, see IRQ_EVT_NNNN (Table 10–3) for a complete list of events. Or, use the PER_get XXX EventId() function to get the EventID.

Return Value

Event flag, 0 or 1

Description

Tests an event to see if its flag is set in the IFR register.

Example

```
while (!IRQ_test(IRQ_EVT_TINT0);
```

Chapter 11

McBSP Module

This chapter describes the structure, functions, and macros of the McBSP module.

Topic **Page**

11.1 Overview 11-2
11.2 Configuration Structure 11-4
11.3 Functions 11-6
11.4 Macros 11-12
11.5 Examples 11-14

11.1 Overview

Table 11–1 summarizes the primary API functions.

- Your application must call MCBSP_open() and MCBSP_close().
- Your application can also call MCBSP_reset(hTimer).
- You can perform configuration by calling either MCBSP_config(), MCBSP_configArgs(), or any of the SET register macros.

  Because MCBSP_config() and MCBSP_configArgs() initialize 27 control registers, macros are provided to enable efficient access to individual registers when you need to set only one or two.

  Using MCBSP_config() to initialize the MCBSP registers is the recommended approach.

The McBSP API defines macros designed for the following primary purposes:

- The *RMK* macros create individual control-register masks for the following purposes:
  - To initialize an MCBSP_Config structure that you then pass to functions such as MCBSP_config().
  - To use as arguments for functions such as MCBSP_configArgs()
  - To use as arguments for the appropriate SET macro.
- Other macros are available primarily to facilitate reading and writing individual bits and fields in the MCBSP control registers.

Table 11–1 (c) lists the most commonly used macros. Section 11.4 includes a description of all McBSP macros.

Table 11–1. McBSP Primary Summary

(a) McBSP Configuration Structure

Structure	Purpose	See page...
MCBSP_Config	McBSP configuration structure used to setup the McBSP interface	11-4

(b) McBSP Functions

Function	Purpose	See page ...
MCBSP_config()	Sets up McBSP using configuration structure (MCBSP_Config)	11-6
MCBSP_configArgs()	Sets up McBSP using register values passed to the function	11-7
MCBSP_getXmtEventID	Retrieves transmit event ID for a given port	11-9
MCBSP_getRcvEventID	Retrieves the IRQ receive event ID for a given port	11-9
MCBSP_open()	Opens the McBSP and assigns a handler to it	11-10
MCBSP_close()	Closes the McBSP and its corresponding handler	11-10
MCBSP_reset()	Resets the McBSP registers with default values	11-11
MCBSP_start()	Starts the McBSP registers with default values	11-11

(c) McBSP Macros

Macro	Purpose	See page ...
MCBSP_XXXX_RMK	Creates a value to store in a particular register (See below)	
MCBSP_RSET(XXXX, Val)	Write a value to store in a particular register (See below)	
MCBSP_RGET(XXXX)	Read a value from a particular register (See below)	
	where XXXX is SPCR1, SPCR2, RCR1, RCR2, XCR1, XCR2, SRGR1, SRGR2, MCR1, MCR2, RCERA, RCERB, RCERC, RCERD, RCERE, RCERF, RCERG, RCERH, XCERA, XCERB, XCERC, XCERD, XCERE, XCERF, XCERG, XCERH, PCR	

11.2 Configuration Structure

This section describes the structure in the McBSP module.

MCBSP_Config — *McBSP configuration structure used to set up McBSP interface*

Structure MCBSP_Config

Members

Uint16 spcrX	(X:1,2)	Serial Port Control Register
Uint16 rcrX	(X:1,2)	Receive Control Register
Uint16 xcrX	(X:1,2)	Transmit Control Register
Uint16 srgrX	(X:1,2)	Sample Rate Generator
Uint16 mcrX	(X:1,2)	Multi–channel Register
Uint16 pcr	Pin Control Register	
Uint16 rcerX	(X:a,b,c,d,e,f,g,h)	Receive Channel Enable Register
Uint16 xcerX	(X:a,b,c,d,e,f,g,h)	Transmit Channel Enable Register

Description The McBSP configuration structure is used to set up the McBSP Interface. You create and initialize this structure and then pass its address to the MCBSP_config() function. You can use literal values or the *MCBSP_RMK* macros to create the structure member values.

Example

```
MCBSP_Config Config1 = {
   0xFFFF, /* spcr1 */
   0x03FF, /* spcr2 */
   0x7FE0, /* rcr1 */
   0xFFFF , /* rcr2.*/
   0x7FE0, /* xcr1 */
   0xFFFF, /* xcr2 */
   0xFFFF, /* srgr1 */
   0xFFFF, /* srgr2 */
   0x03FF, /* mcr1 */
   0x03FF , /* mcr2.*/
   0xFFFF   /* pcr*/
   0xFFFF, /* rcera */
   0xFFFF, /* rcerb */
   0xFFFF, /* rcecrc */
   0xFFFF , /* rcerd.*/
   0xFFFF, /* rcere */
   0xFFFF, /* rcerf */
   0xFFFF, /* rcecrg */
```

```
    0xFFFF , /* rcerh.*/
    0xFFFF, /* xcera */
    0xFFFF, /* xcerb */
    0xFFFF, /* xcecrc */
    0xFFFF , /* xcerd.*/
    0xFFFF, /* xcere */
    0xFFFF, /* xcerf */
    0xFFFF, /* xcecrg */
    0xFFFF  /* xcerh*/
}
```

11.3 Functions

This section describes the functions in the McBSP module.

MCBSP_config *Writes value to up McBSP using configuration structure*

Function

void MCBSP_config(MCBSP_Handle hMcbsp,
    MCBSP_Config *Config
);

Arguments

hMcbsp	Device handler
Config	Pointer to an initialized configuration structure

Return Value None

Description Writes a value to up the McBSP using the configuration structure. The values of the structure are written to the port registers (see also MCBSP_configArgs() and MCBSP_Config).

Example

```
MCBSP_Config MyConfig = {
   0xFFFF, /* spcr1 */
   0x03FF, /* spcr2 */
   0x7FE0, /* rcr1 */
   0xFFFF, /* rcr2 */
   0x7FE0, /* xcr1 */
   0xFFFF, /* xcr2 */
   0xFFFF, /* srgr1 */
   0xFFFF, /* srgr2 */
   0x03FF, /* mcr1 */
   0x03FF, /* mcr2 */
   0xFFFF, /* pcr */
   0xFFFF, /* rcera */
   0xFFFF, /* rcerb */
   0xFFFF, /* rcecrc */
   0xFFFF , /* rcerd */
   0xFFFF, /* rcere */
   0xFFFF, /* rcerf */
   0xFFFF, /* rcecrg */
   0xFFFF, /* rcerh */
   0xFFFF, /* xcera */
   0xFFFF, /* xcerb */
   0xFFFF, /* xcecrc */
   0xFFFF, /* xcerd */
   0xFFFF, /* xcere */
   0xFFFF, /* xcerf */
   0xFFFF, /* xcecrg */
   0xFFFF, /* xcerh */

   MCBSP_config(myhMcbsp, &MyConfig);
```

MCBSP_configArgs *Writes to McBSP using register values passed to function*

Function

```
void MCBSP_configArgs(hMcbsp,
    0xFFFF, /* spcr1 */
    0x03FF, /* spcr2 */
    0x7FE0, /* rcr1 */
    0xFFFF, /* rcr2.*/
    0x7FE0, /* xcr1 */
    0xFFFF, /* xcr2 */
    0xFFFF, /* srgr1 */
    0xFFFF, /* srgr2 */
    0x03FF, /* mcr1 */
    0x03FF, /* mcr2.*/
    0xFFFF, /* pcr */
    0xFFFF, /* rcera */
    0xFFFF, /* rcerb */
    0xFFFF, /* rcecrc */
    0xFFFF, /* rcerd.*/
    0xFFFF, /* rcere */
    0xFFFF, /* rcerf */
    0xFFFF, /* rcecrg */
    0xFFFF, /* rcerh.*/
    0xFFFF, /* xcera */
    0xFFFF, /* xcerb */
    0xFFFF, /* xcecrc */
    0xFFFF, /* xcerd.*/
    0xFFFF, /* xcere */
    0xFFFF, /* xcerf */
    0xFFFF, /* xcecrg */
    0xFFFF, /* xcerh.*/
);
```

Arguments

MCBSP_Handle		HMCBSP Device handler
Uint16 spcrX	(X:1,2)	Serial Port Control Register
Uint16 rcrX	(X:1,2)	Receive Control Register
Uint16 xcrX	(X:1,2)	Transmit Control Register
Uint16 srgrX	(X:1,2)	Sample Rate Generator
Uint16 mcrX	(X:1,2)	Multi–channel Register
Uint16 pcr		Pin Control Register
Uint16 rcerX	(X:a,b,c,d,e,f,g,h)	Receive Channel Enable Register
Uint16 xcerX	(X:a,b,c,d,e,f,g,h)	Transmit Channel Enable Register

Return Value None

Description Writes to the McBSP using the register values passed to the function. The register values are written to the McBSP registers.

You may use literal values for the arguments; or for readability, you may use the MCBSP_RMK macros to create the register values based on field values.

Example

```
MCBSP_configArgs (
   0xFFFF, /* spcr1 */
   0x03FF, /* spcr2 */
   0x7FE0, /* rcr1 */
   0xFFFF, /* rcr2 */
   0x7FE0, /* xcr1 */
   0xFFFF, /* xcr2 */
   0xFFFF, /* srgr1 */
   0xFFFF, /* srgr2 */
   0x03FF, /* mcr1 */
   0x03FF, /* mcr2 */
   0xFFFF, /* pcr */
   0xFFFF, /* rcera */
   0xFFFF, /* rcerb */
   0xFFFF, /* rcecrc */
   0xFFFF, /* rcerd */
   0xFFFF, /* rcere */
   0xFFFF, /* rcerf */
   0xFFFF, /* rcecrg */
   0xFFFF, /* rcerh.*/
   0xFFFF, /* xcera */
   0xFFFF, /* xcerb */
   0xFFFF, /* xcecrc */
   0xFFFF, /* xcerd */
   0xFFFF, /* xcere */
   0xFFFF, /* xcerf */
   0xFFFF, /* xcecrg */
   0xFFFF /* xcerh */
);
```

MCBSP_getXmtEventID

Retrieves transmit event ID for given port

Function

```
Uint16 MCBSP_getXmtEventId(
    MCBSP_Handle hMcbsp
);
```

Arguments

hMcbsp   Handle to McBSP port obtained by MCBSP_open()

Return Value

Receiver event ID

Description

Simple replace receive for transmit. Use this ID to manage the event using the IRQ module.

Example

```
Uint16 XmtEventId;
...
XmtEventId = MCBSP_getXmtEventId(hMcbsp);
IRQ_enable(XmtEventId);
```

MCBSP_getRcvEventId

Retrieves receive event ID for a given port

Function

```
Uint16 MCBSP_getRcvEventId(
    MCBSP_Handle hMcbsp
);
```

Arguments

hMcbsp   Handle to McBSP port obtained by MCBSP_open()

Return Value

Receiver event ID

Description

Retrieves the IRQ receive event ID for a given port. Use this ID to manage the event using the IRQ module.

Example

```
Uint16 RecvEventId;
...
RecvEventId = MCBSP_getRcvEventId(hMcbsp);
IRQ_enable(RecvEventId);
```

MCBSP_open

Opens McBSP for McBSP calls

Function

MCBSP_Handle MCBSP_open(
  int devnum
  Uint32 flags
);

Arguments

devnum	Mcbsp Device Number: MCBSP_PORT0, MCBSP_PORT1, MCBSP_PORT2
flags	Event Flag Number: Logical open or MCBSP_OPEN_RESET

Return Value

MCBSP_Handle   Device handler

Description

Before a McBSP device can be used, it must first be opened by this function. Once opened, it cannot be opened again until closed (see MCBSP_close). The return value is a unique device handle that is used in subsequent McBSP API calls. If the function fails, INV is returned. If the MCBSP_OPEN_RESET is specified, then the power on defaults are set and any interupts are disabled and cleared.

Example

```
Mcbsp_Handle hMcbsp;
...
hMcbsp = MCBSP_open(MCBSP_PORT0,0);
```

MCBSP_close

Closes McBSP

Function

void MCBSP_close
  MCBSP_Handle hMcbsp
);

Arguments

hMcbsp   Device Handle (see MCBSP_open()).

Return Value

MCBSP_Handle   Device handler

Description

Closes a previously opened McBSP device. The McBSP event is disabled and cleared. The McBSP registers are set to their default values.

Example

```
Mcbsp_close(hMcbsp);
```

MCBSP_reset — *Resets McBSP*

Function

void MCBSP_reset
MCBSP_Handle hMcbsp
);

Arguments

hMcbsp Device handle, see MCBSP_open();

Return Value

None

Description

Resets the McBSP device. Disables and clears the interrupt event and sets the McBSP registers to default values. If INV is specified, all McBSP devices are reset.

Example

```
Mcbsp_reset(hMcbsp);
```

MCBSP_start — *Starts transmit and/or receive operation for a MCBSP port*

Function

void MCBSP_start(
MCBSP_Handle hMcbsp,
Uint16 txRxSelectorstartMask,
Uint16 SampleRaterateGenDelay
);

Arguments

hMcbsp Handle to McBSP port obtained by MCBSP_open()
txRxSelector Start transmit, receive or both:

- MCBSP_XMIT_START
- MCBSP_RCV_START
- MCBSP_XMIT_START | MCBSP_RCV_START

SampleRateGenDelay Sample rate generates delay. MCBSP logic requires two sample_rate generator clock_periods after grabbing the sample rate generator logic to stabilize. Use this parameter to provide the appropriate delay before starting the MCBSP. A conservative value should be equal to:

$$SampleRateGenDelay = \frac{2\ x\ Sample\_Rate\_Generator\_Clock\_period}{4xC54x\_Instruction\_Cycle}$$

Return Value

None

Description

Starts a transmit and/or receive operation for a MCBSP port.

Example

```
Mcbsp_start(hMcbsp);
```

11.4 Macros

Table 11–2. MCBSP Macros Using MCBSP Port Number

(a) Macros to read/write MCBSP register values

Macro	Syntax
MCBSP_RGET()	Uint16 MCBSP_RGET(*REG*)
MCBSP_RSET()	Void MCBSP_RSET(*REG*, Uint16 *regval*)

(b) Macros to read/write MCBSP register field values (Applicable only to registers with more than one field)

Macro	Syntax
MCBSP_FGET()	Uint16 MCBSP_FGET(*REG*, *FIELD*)
MCBSP_FSET()	Void MCBSP_FSET(*REG*, *FIELD*, Uint16 *fieldval*)

(c) Macros to create value to MCBSP registers and fields (Applies only to registers with more than one field)

Macro	Syntax
MCBSP_REG_RMK()	Uint16 MCBSP_REG_RMK(*fieldval_n*,…*fieldval_0*) Note: *Start with field values with most significant field positions: field_n: MSB field field_0: LSB field *only writable fields allowed
MCBSP_FMK()	Uint16 MCBSP_FMK(*REG*, *FIELD*, *fieldval*)

(d) Macros to read a register address

Macro	Syntax
MCBSP_ADDR()	Uint16 MCBSP_ADDR(*REG*)

Notes:
1) *REG* indicates the register, xxx xxx.
2) *FIELD* indicates the register field name as specified in Appendix A.
   - ❑ For *REG*_FSET and *REG*_FMK, *FIELD* must be a writable field.
   - ❑ For *REG*_FGET, the field must be a writable field.
3) *regval* indicates the value to write in the register (*REG*).
4) *fieldval* indicates the value to write in the field (*FIELD*).

Table 11–3. McBSP CSL Macros Using Handle

(a) Macros to read/write MCBSP register values

Macro	Syntax
MCBSP_RGET_H()	Uint16 MCBSP_RGET_H(MCBSP_Handle hMCBSP, *REG*)
MCBSP_RSET_H()	Void MCBSP_RSET_H(MCBSP_Handle hMCBSP, *REG*, Uint16 *regval*)

(b) Macros to read/write MCBSP register field values (Applicable only to registers with more than one field)

Macro	Syntax
MCBSP_FGET_H()	Uint16 MCBSP_FGET_H(MCBSP_Handle hMCBSP, *REG*, *FIELD*)
MCBSP_FSET_H()	Void MCBSP_FSET_H(MCBSP_Handle hMCBSP, *REG*, *FIELD*, *Uint16 fieldval*)

(c) Macros to read a register address

Macro	Syntax
MCBSP_ADDR_H()	Uint16 MCBSP_ADDR_H(MCBSP_Handle hMCBSP, *REG*)

Notes:
1) *REG* indicates the register, xxx xxx.
2) *FIELD* indicates the register field name as specified in Appendix A.
   - ❑ For *REG*_FSET_H, *FIELD* must be a writable field.
   - ❑ For *REG*_FGET, the field must be a writable field.
3) *regval* indicates the value to write in the register (*REG*).
4) *fieldval* indicates the value to write in the field (*FIELD*).

11.5 Examples

The following CSL McBSP initialization examples are provided under the \examples\MCBSP directory.

Example 11–1 illustrates the McBSP port initialization using MCBSP_config(). The example also explains how to set the McBSP into digital loopback mode and perform 32-bit reads/writes from/to the serial port.

Example 11–1. McBSP Port Initialization Using MCBSP_config()

```
#include <csl_mcbsp.h>

static MCBSP_Config ConfigLoopBack32= {
   ....
};


void main(void) {

  MCBSP_Handle mhMcbsp;
  Uint32 xmt, rcv;

  ....
   CSL_init();

   mhMcbsp = MCBSP_open(MCBSP_DEV0, MCBSP_OPEN_RESET);

   MCBSP_config(mhMcbsp, &ConfigLoopBack32);

   MCBSP_start(mhMcbsp,MCBSP_RCV_START|MCBSP_XMIT_START|
   MCBSP_SRGR_START|MCBSP_SRGR_FRAMESYNC,0x300u);

......

   while (!MCBSP_FGET_H(mhMcbsp,SPCR2,XRDY));
   MCBSP_write32(mhMcbsp,xmt[i]);

......

   while (!MCBSP_FGET_H(mhMcbsp,SPCR1,RRDY));
   rcv[i] = MCBSP_read32(mhMcbsp);

......

   MCBSP_close(mhMcbsp);
```

Chapter 12

PLL Module

This chapter describes the structure, functions, and macros of the PLL module.

Topic **Page**

12.1 Overview **12-2**

12.2 Configuration Structure **12-4**

12.3 Functions **12-5**

12.4 Macros **12-8**

12.1 Overview

Table 12–1 summarizes the primary API functions. A shaded row indicates functions required to control the pll interface through the CSL.

- Your application must call PLL_open(), and PLL_close() in order to access the different PLL functions and macros.
- You can perform configuration by calling either PLL_config(), PLL_configArgs(), or any of the SET register macros.

  Using PLL_config() to initialize the PLL registers is the recommended approach.

The PLL API defines macros designed for the following primary purposes:

- The RMK macros create individual control-register masks for the following purposes:
  - To initialize an PLL_Config structure that you then pass to functions such as PLL_config().
  - To use as arguments for functions such as PLL_configArgs()..
  - To use as arguments for the appropriate SET macro.
- Other macros are available primarily to facilitate reading and writing individual bits and fields in the PLL control registers.

Table 12–1 (c) lists the most commonly used macros. Section 12.4 includes a description of all PLL macros.

Table 12–1. PLL Primary Summary

(a) PLL Configuration Structure

Structure	Purpose	See page...
PLL_Config	PLL configuration structure used to setup the PLL interface	12-4

(b) PLL Functions

Function	Purpose	See page ...
PLL_config()	Sets up PLL using configuration structure (PLL_Config)	12-6
PLL_configArgs()	Sets up PLL using register values passed to the function	12-6
PLL_open()	Opens the PLL for PLL register changes and assigns a PLL handle	12-5
PLL_close	Closes the PLL and the corresponding handler	12-5

(c) PLL Macros

Macro	Purpose	See page ...
PLL_XXXX_RMK	Creates a value to store in a particular register (See below)	
PLL_RSET (XXXX, Val)	Write a value to store in a particular register (See below)	
PLL_RGET(XXXX)	Read a value from a particular register (See below)	
	where XXXX is CLKMD	

12.2 Configuration Structure

This section describes the structure in the PLL module.

PLL_Config — *PLL configuration structure used to set up PLL interface*

Structure PLL_Config

Members

Uint16 iai	Initialize After Idle
Uint16 iob	Initialize On Break
Uint16 pllmult	PLL Multiply value
Uint16 plldiv	PLL Divide value

Description The PLL configuration structure is used to set up the PLL Interface. You create and initialize this structure and then pass its address to the PLL_config() function. You can use literal values or the *PLL_RMK* macros to create the structure member values.

Example

```
PLL_Config Config1 = {
   1,      /* iai */
   1,      /* iob */
   31,     /* pllmult */
   3       /* plldiv */
}
```

12.3 Functions

This section describes the functions in the PLL module.

PLL_open — *Opens PLL API for PLL API calls*

Function

PLL_Handle PLL_open(
    bool clkmd1
    Uint16 flags
);

Arguments

Clkmd1	Clock Mode Number: 1 or 0
flags	Event Flag Number: Logical open, or PLL_OPEN_RESET

Return Value

PLL_Handle	Device handler

Description

Before a PLL device can be used, it must first be opened by this function. Once opened, it cannot be opened again until closed (see PLL_close). The return value is a unique device handle that is used in subsequent PLL API calls. If the function fails, INV is returned. If the PLL_OPEN_RESET is specified, then the power on defaults are set and any interrupts are disabled and cleared.

Example

```
Pll_Handle hPll;
hPll = PLL_open(0,0);
```

PLL_close — *Closes PLL*

Function

void PLL_close
    PLL_Handle hPll
);

Arguments

hPll	Device Handle (see PLL_open());

Return Value

PLL_Handle	Device handler

Description

Closes a previously opened pll device. The pll event is disabled and cleared. The clkmd register is set to its default value.

Example

```
Pll_close(hPll);
```

PLL_config — *Writes value to up PLL using configuration structure*

Function

void PLL_config(PLL_Handle hPll,
    PLL_Config *Config
);

Arguments

Config	Pointer to an initialized configuration structure
hPll	Device handle (see PLL_open()).

Return Value

None

Description

Writes a value to up the PLL using the configuration structure. The values of the structure are written to the port registers (see also PLL_configArgs() and PLL_Config).

Example

```
PLL_Config MyConfig = {
   1,      /* iai */
   1,      /* iab */
   31,     /* pllmult */
   3       /* plldiv */
}
PLL_config(hPll&MyConfig);
```

PLL_configArgs — *Writes to PLL using register values passed to function*

Function

void PLL_configArgs(PLL_Handle hPll,
Uint16 iai,
Uint16 iob,
Uint16 pllmult,
Uint16 plldiv)

Arguments

hPll	Device handle (see PLL_open()).
iai	Initialize After Idle
iob	Initialize On Break
pllmult	PLL Multiply value
plldiv	PLL Divide value

Return Value

None

Description

Writes to the PLL using the register values passed to the function. The register values are written to the PLL registers.

You may use literal values for the arguments; or for readability, you may use the PLL_RMK macros to create the register values based on field values.

Example

```
PLL_configArgs (hPll,
   1,  /* iai */
   1,  /* iob */
   31, /* pllmult */
   3,  /* plldiv */
 )
```

12.4 Macros

CSL offers a collection of macros to gain individual access to the PLL peripheral registers and fields..

Table 12–2 contains a list of macros available for the PLL module. To use them, include ”csl_pll.h”.

Table 12–2. PLL CSL Macros Using PLL Port Number

(a) Macros to read/write PLL register values

Macro	Syntax
PLL_RGET()	Uint16 PLL_RGET(*REG*)
PLL_RSET()	Void PLL_RSET(*REG*, Uint16 *regval*)

(b) Macros to read/write PLL register field values (Applicable only to registers with more than one field)

Macro	Syntax
PLL_FGET()	Uint16 PLL_FGET(*REG*, *FIELD*)
PLL_FSET()	Void PLL_FSET(*REG*, *FIELD*, Uint16 *fieldval*)

(c) Macros to create value to PLL registers and fields (Applies only to registers with more than one field)

Macro	Syntax
PLL_REG_RMK()	Uint16 PLL_REG_RMK(*fieldval_n*,…*fieldval_0*) Note: *Start with field values with most significant field positions: field_n: MSB field field_0: LSB field *only writable fields allowed
PLL_FMK()	Uint16 PLL_FMK(*REG*, *FIELD*, *fieldval*)

(d) Macros to read a register address

Macro	Syntax
PLL_ADDR()	Uint16 PLL_ADDR(*REG*)

Notes: 1) *REG* indicates the register, xxx xxx.

2) *FIELD* indicates the register field name as specified in Appendix A.
   - ❑ For *REG*_FSET and *REG*_FMK, *FIELD* must be a writable field.
   - ❑ For *REG*_FGET, the field must be a writable field.

3) *regval* indicates the value to write in the register (*REG*).

4) *fieldval* indicates the value to write in the field (*FIELD*).

Chapter 13

PWR Module

The CSL PWR module offers functions to control the power consumption of different sections in the C54x device.

Topic **Page**

13.1 Overview **13-2**

13.2 Functions **13-3**

13.3 Macros **13-4**

13.1 Overview

The CSL PWR module offers functions to control the power consumption of different sections in the C54x device. The PWR module is not handle-based.

Currently, there are no macros available for the power-down module.

Table 13–1 lists the functions for use with the PWR modules that order specific parts of the C54x to power down.

Table 13–1. PWR Functions

Function	Purpose	See page ...
PWR_powerDown	Forces the DSP to enter a power-down state	13-3

13.2 Functions

This section lists the functions in the PWR module.

PWR_powerDown *Forces DSP to enter power-down state*

Function void PWR_powerDown (Uint16 powerDownMode, Uint16 wakeUpMode)

Arguments powerDownMode : Mask for ICR register
wakeMode:

- _WAKEUP_MI_ : PWR_WAKEUP_MI_ wakes up with an unmasked interrupt and jump to execute the ISR's executed.
- _WAKEUP_NMI : PWR_WAKEUP_MI_ wakes up with an unmasked interrupt and executes the next following instruction (interrupt is not take).

Return Value None

Description Power-down the device in different power-down and wake-up modes. In the C54x, power-down is acheived by executing an IDLE K instruction.

Example

```
PWR_powerDown (1, PWR_WAKEUP_MI);
```

13.3 Macros

CSL offers a collection of macros to gain individual access to the PWR peripheral registers and fields..

Table 13–2 contains a list of macros available for the PWR module. To use them, include ”csl_pwr.h”.

Table 13–2. PWR CSL Macros Using PWR Port Number

(a) Macros to read/write PWR register values

Macro	Syntax
PWR_RGET()	Uint16 PWR_RGET(*REG*)
PWR_RSET()	Void PWR_RSET(*REG*, Uint16 *regval*)

(b) Macros to read/write PWR register field values (Applicable only to registers with more than one field)

Macro	Syntax
PWR_FGET()	Uint16 PWR_FGET(*REG*, *FIELD*)
PWR_FSET()	Void PWR_FSET(*REG*, *FIELD*, Uint16 *fieldval*)

(c) Macros to create value to PWR registers and fields (Applies only to registers with more than one field)

Macro	Syntax
PWR_REG_RMK()	Uint16 PWR_REG_RMK(*fieldval_n*,…*fieldval_0*) Note: *Start with field values with most significant field positions: field_n: MSB field field_0: LSB field *only writable fields allowed
PWR_FMK()	Uint16 PWR_FMK(*REG*, *FIELD*, *fieldval*)

(d) Macros to read a register address

Macro	Syntax
PWR_ADDR()	Uint16 PWR_ADDR(*REG*)

Notes: 1) *REG* indicates the register, xxx xxx. ICR, ISTR

2) *FIELD* indicates the register field name as specified in Appendix A.
   - ❑ For *REG*_FSET and *REG*_FMK, *FIELD* must be a writeable field.
   - ❑ For *REG*_FGET, the field must be a writable field.

3) *regval* indicates the value to write in the register (*REG*).

4) *fieldval* indicates the value to write in the field (*FIELD*).

Chapter 14

TIMER Module

This chapter describes the structure, functions, and macros of the TIMER module.

Topic **Page**

14.1 Overview 14-2

14.2 Configuration Structure 14-4

14.3 Functions 14-5

14.4 Macros 14-9

14.1 Overview

Table 14–1 (a) summarizes the primary timer functions.

- Your application must call TIMER_open() and TIMER_close().
- Your application can also call TIMER_reset.
- You can perform configuration by calling either TIMER_config(), TIMER_configArgs(), or any of the TIMER_SET register macros.

  Because TIMER_config() and TIMER_configArgs() initialize all four control registers, macros are provided to enable efficient access to individual registers when you need to set only one or two.

  Using TIMER_config() to initialize the timer registers is the recommended approach.

The TIMER API defines macros designed for the following primary purposes:

- The *RMK* macros create individual control-register masks for the following purposes:
  - To initialize an TIMER_Config structure that you then pass to functions such as TIMER_config().
  - To use as arguments for functions such as TIMER_configArgs()
  - To use as arguments for the appropriate SET macro.
- Other macros are available primarily to facilitate reading and writing individual bits and fields in the TIMER control registers.

Table 14–1(c) lists the most commonly used macros. Section 14.4 includes a description of all TIMER macros.

Table 14–1. TIMER Primary Summary

(a) TIMER Configuration Structure

Structure	Purpose	See page...
TIMER_Config	TIMER configuration structure used to setup the TIMER_config() function	14-4

(b) TIMER Functions

Function	Purpose	See page ...
TIMER_config()	Sets up TIMER using configuration structure (TIMER_Config)	14-5
TIMER_configArgs()	Sets up TIMER using register values passed to the function	14-5
TIMER_getEventId	Obtains IRQ event ID for TIMER device	14-6
TIMER_tintoutCfg()	Sets up the TIMER Polarity pin along with settings for the FUNC, PWID, CP fields in the TCR register	14-8
TIMER_open()	Opens the TIMER and assigns a handler to it	14-6
TIMER_close()	Closes the TIMER and its corresponding handler	14-7
TIMER_reset()	Resets the TIMER registers with default values	14-7
TIMER_start()	Starts the TIMER device running	14-7
TIMER_stop()	Stops the TIMER device running	14-8

(c) TIMER Macros

Macro	Purpose	See page ...
TIMER_XXXX_RMK	Creates a value to store in a particular register (See below)	14-10
TIMER_RSET(XXXX, Val)	Write a value to store in a particular register (See below)	14-10
TIMER_RGET(XXXX)	Read a value from a particular register (See below) where XXXX is TCR, PRD, TIM, or PRSC	14-10

14.2 Configuration Structure

This section describes the structure in the TIMER module.

TIMER_Config — *TIMER configuration structure*

Structure

TIMER_Config

Members

Uint16 tcr	Timer Control Register
Uint16 prd	Period Register
Uint16 prsc	Timer Pre-scaler Register

Description

The TIMER configuration structure is used to setup a timer device. You create and initialize this structure then pass its address to the TIMER_config() function. You can use literal values or the TIMER_RMK macros to create the structure member values.

Example

```
TIMER_Config Config1 = {
   0x0010, /* tcr */
   0xFFFF, /* prd */
   0xF0F0, /* prsc */
}
```

14.3 Functions

This section describes the functions in the TIMER module.

TIMER_config — *Writes value to TIMER using configuration structure*

Function

void TIMER_config(TIMER_Handle hTimer,
    TIMER_Config *Config
);

Arguments

Config	Pointer to an initialized configuration structure
hTimer	Device Handle, see TIMER_open

Return Value

none

Description

The values of the configuration structure are written to the timer registers (see also TIMER_configArgs() and TIMER_Config).

Example

```
TIMER_Config MyConfig = {
   0x0010, /* tcr */
   0xFFFF, /* prd */
   0xF0F0, /* prsc */
}
   TIMER_config(hTimer&MyConfig);
```

TIMER_configArgs — *Writes to TIMER register values passed to function*

Function

void TIMER_configArgs(`Timer_Handle hTimer`
    Uint16 tcr
    Uint16 prd
    Uint16 prsc
);

Arguments

hTimer	Device Handler (see TIMER_open).
prd	Period Register
tcr	Timer Control Register
prsc	Timer Prescaler Register

Return Value

none

Description

Writes to the TIMER register values passed to the function.

You may use literal values for the arguments; or for readability, you may use the TIMER_REG_RMK macros to create the register values based on field values.

Example

```
TIMER_configArgs (hTimer,
   0xFFFF, /* tcr */
   0xFFFF, /* prd */
   0x0010, /* prsc */
);
```

TIMER_getEventId *Obtains IRQ event ID for TIMER device*

Function

Uint16 TIMER_getEventId(
    TIMER_Handle hTimer
);

Arguments

hTimer	Device handle (see TIMER_open).

Return Value

Event ID	IRQ Event ID for the timer device

Description

Obtains the IRQ event ID for the timer device (see IRQ Module in Chapter 10-10).

Example

```
TimerEventId = TIMER_getEventId(hTimer);
IRQ_enable(TimerEventId);
```

TIMER_open *Opens TIMER for TIMER calls*

Function

TIMER_Handle TIMER_open(
    int devnum
    Uint16 flags
);

Arguments

devnum	Timer Device Number: TIMER_DEV0, TIMER_DEV1
flags	Event Flag Number: Logical open or TIMER_OPEN_RESET

Return Value

TIMER_Handle	Device handler

Description

Before a TIMER device can be used, it must first be opened by this function. Once opened, it cannot be opened again until closed, see TIMER_close. The return value is a unique device handle that is used in subsequent TIMER calls. If the function fails, an INV (–I) value is returned. If the TIMER_OPEN_RESET is specified, then the power on defaults are set and any interrupts are disabled and cleared.

Example

```
Timer_Handle hTimer;
...
hTimer = TIMER_open(TIMER_DEV0,0);
```

TIMER_close — *Closes TIMER*

Function
void TIMER_close
    TIMER_Handle hTimer
 );

Arguments hTimer   Device Handle (see TIMER_open).

Return Value TIMER_Handle Device handler

Description Closes a previously opened timer device. The timer event is disabled and cleared. The timer registers are set to their default values.

Example
```
Timer_close(hTimer);
```

TIMER_reset — *Resets TIMER*

Function
void TIMER_reset
    TIMER_Handle hTimer
 );

Arguments hTimer   Device handle (see TIMER_open).

Return Value none

Description Resets the timer device. Disables and clears the interrupt event and sets the timer registers to default values. If INV (–I) is specified, all timer devices are reset.

Example
```
Timer_reset(hTimer);
```

TIMER_start — *Starts TIMER device running*

Function
void TIMER_start(
    TIMER_Handle hTimer
);

Arguments hTimer   Device handle (see TIMER_open).

Return Value none

Description Starts the timer device running. TSS field =0.

Example
```
TIMER_start(hTimer);
```

TIMER_stop — *Stops TIMER device running*

Function

```
void TIMER_stop(
    TIMER_Handle hTimer
);
```

Arguments

hTimer	Device handle (see TIMER_open).

Return Value

none

Description

Stops the timer device running. TSS field =1.

Example

```
TIMER_stop(hTimer);
```

TIMER_tintoutCfg — *Configures TINT/TOUT pin*

Function

```
void TIMER_tintoutCfg
    Uint16 idleen,
    TIMER_Handle hTimer,
    Uint16 func,
    Uint16 pwid,
    Uint16 cp,
    Uint16 polar
 );
```

Arguments

hTimer	Device handle (see TIMER_open).
idleen	
func	Function of the TIN/TOUT pin and the source of the timer module.
pwid	TIN/TOUT pulse width
cp	Clock or pulse mode
polar	Polarity of the TIN/TOUT pin

Return Value

none

Description

Configures the TIN/TOUT pin of the device using the TCR register

Example

```
Timer_tintoutCfg(hTimer,
1, /*idleen*/
1, /*funct*/
0, /*pwid*/
0, /*cp*/
0  /*polar*/ );
```

14.4 Macros

CSL offers a collection of macros to gain individual access to the TIMER peripheral registers and fields..

Table 14–2 contains a list of macros available for the TIMER module. To use them, include ”csl_timer.h”.

Table 14–2. TIMER CSL Macros Using Timer Port Number

(a) Macros to read/write TIMER register values

Macro	Syntax
TIMER_RGET()	Uint16 TIMER_RGET(*REG*)
TIMER_RSET()	Void TIMER_RSET(*REG*, Uint16 *regval*)

(b) Macros to read/write TIMER register field values (Applicable only to registers with more than one field)

Macro	Syntax
TIMER_FGET()	Uint16 TIMER_FGET(*REG*, *FIELD*)
TIMER_FSET()	Void TIMER_FSET(*REG*, *FIELD*, Uint16 *fieldval*)

(c) Macros to create value to TIMER registers and fields (Applies only to registers with more than one field)

Macro	Syntax
TIMER_REG_RMK()	Uint16 TIMER_REG_RMK(*fieldval_n*,…*fieldval_0*) Note: *Start with field values with most significant field positions: field_n: MSB field field_0: LSB field *only writable fields allowed
TIMER_FMK()	Uint16 TIMER_FMK(*REG*, *FIELD*, *fieldval*)

(d) Macros to read a register address

Macro	Syntax
TIMER_ADDR()	Uint16 TIMER_ADDR(*REG*)

Notes:
1) *REG* indicates the register, xxx xxx.
2) *FIELD* indicates the register field name as specified in Appendix xxx.
   - ❑ For *REG*_FSET and *REG*_FMK, *FIELD* must be a writable field.
   - ❑ For *REG*_FGET, the field must be a writable field.
3) *regval* indicates the value to write in the register (*REG*).
4) *fieldval* indicates the value to write in the field (*FIELD*).

Table 14–3. TIMER CSL Macros Using Handle

(a) Macros to read/write TIMER register values

Macro	Syntax
TIMER_RGET_H()	Uint16 TIMER_RGET_H(TIMER_Handle hTimer, *REG*)
TIMER_RSET_H()	Void TIMER_RSET_H(TIMER_Handle hTimer, *REG*, Uint16 *regval*)

(b) Macros to read/write TIMER register field values (Applicable only to registers with more than one field)

Macro	Syntax
TIMER_FGET_H()	Uint16 TIMER_FGET_H(TIMER_Handle hTimer, *REG*, *FIELD*)
TIMER_FSET_H()	Void TIMER_FSET_H(TIMER_Handle hTimer, *REG*, *FIELD*, *Uint16 fieldval*)

(c) Macros to read a register address

Macro	Syntax
TIMER_ADDR_H()	Uint16 TIMER_ADDR_H(TIMER_Handle hTimer, *REG*)

Notes:
1) *REG* indicates the register, xxx xxx.
2) *FIELD* indicates the register field name as specified in Appendix A.
   - ❏ For *REG*_FSET_H, *FIELD* must be a writable field.
   - ❏ For *REG*_FGET_H, the field must be a writable field.
3) *regval* indicates the value to write in the register (*REG*).
4) *fieldval* indicates the value to write in the field (*FIELD*).

Index

A

Advanced A and B Pages, 3-9
API Modules, Illustration of, 1-3
Autoinit, 3-9

B

Body section, 2-10
Bool, description of, 1-6
Build Options, 2-25
  Adding the Include Search Path, 2-28
  Defining a Target Device, 2-26
  Defining Large Memory Model, 2-27
  Defining Library Paths, 2-29

C

C Code Generation for CACHE Module, 3-6
  Header File, 3-6
  Source File, 3-7
C Code Generation for DMA Module, 3-12
  Header File, 3-12
  Source File, 3-13
C Code Generation for EMIF Module, 3-18
  Header File, 3-18
  Source File (Declaration Section), 3-18
C Code Generation for GPIO Module, 3-21
  Header File, 3-21
  Source File, 3-21
C Code Generation for MCBSP Module, 3-27
  Header File, 3-27
  Source File, 3-28
C Code Generation for PLL Module, 3-33
  Header File, 3-33
  Source File, 3-33
C Code Generation for TIMER, 3-39
C Code Generation for TIMER Module
  Header File, 3-39
  Source File, 3-40
C File, mandatory include, 2-8
CACHE
  Configuration Manager, 3-3
  Configuration Structure, 4-3
  Functions, 4-3
  Macros, 4-3
  overview, 4-2
  Primary Summary, 4-3
CACHE Configuration Manager, 3-3
  Configuring the Object Properties, 3-4
  Creating/Inserting a Configuration, 3-3
  Delete/Rename Object, 3-3
CACHE Configuration Structure, CACHE_Config, 4-3, 4-4
CACHE Functions
  CACHE_config, 4-3, 4-5
  CACHE_configArgs, 4-3, 4-6
  CACHE_enable, 4-3, 4-7
  CACHE_flush, 4-3, 4-7
CACHE Header File, Example of, 3-6
CACHE Macros
  CACHE_ADDR, 4-8
  CACHE_FGET, 4-8
  CACHE_FMK, 4-8
  CACHE_FSET, 4-8
  CACHE_REG_RMK, 4-8
  CACHE_RGET, 4-8
  CACHE_RGET_GET, 4-3
  CACHE_RSET, 4-8
  CACHE_RSET_SET, 4-3
  CACHE_XXXX_RMK, 4-3
  Using Port Number, 4-8, 13-4

CACHE Module
C Code Generation for, 3-6
overview, 3-3
CACHE Properties Page
illustration of, 3-4
With Handle Object Accessible, , illustration of, 3-6
CACHE Resource Manager, 3-3, 3-5
Properties Page, 3-5
CACHE Resource Manager Menu, illustration of, 3-5
CACHE Sections Menu, illustration of, 3-3
CACHE Source File (Body Section), example of, 3-7
CACHE Source File (Declaration Section), example of, 3-7
CACHE_config, 3-7, 4-2
CACHE_configArgs, 4-2
CACHE_flush, 4-2
CACHE_open, 3-7
CACHE_open(), 3-7, 3-13
CHIP, overview, 5-2
CHIP , functions, 5-2
CHIP Functions
CHIP_getDieID_High32, 5-2, 5-3
CHIP_getDieID_Low32, 5-2, 5-4
CHIP_getEndian, 5-2, 5-3
CHIP_getRevID, 5-2, 5-3
CHIP_getSubsysId, 5-4
Chip Support Library, 1-2, 3-1
chip support library (CSL)
naming conventions, 1-5
notational conventions, iv
Code Composer Studio, 3-1
Adding the Include Search Path, 2-28
Defining a Target Device, 2-26
Defining Large Memory Model, 2-27
Defining Library Paths, 2-29
Installing, 2-2
using the CCS Project Environment, 2-25
Compiling, with CSL, 2-24
Configuration Structure
CACHE, 4-4
DMA, 7-4
EMIF, 8-5
IRQ, 10-7
MCBSP, 11-4
TIMER, 14-4
Configuration Structure, PLL, 12-4
Configuring Peripherals Without GUI, 2-20
Constant Values for Fields, 1-8
Constant Values for Registers, 1-8
CSL
Architecture, 1-2
Compiling and Linking with, 2-24
Data Types, 1-6
Destination Address, 2-20
Device Support, 1-4
Directory Structure, 2-24
Introduction to, 1-2
Linking with, 2-24
Macros, generic, 1-9
Handle–based, 1-10
Modules and Include Files, 1-3
Naming Conventions, 1-5
Rebuilding CSL, 2-31
Resource Management, 1-6
Source Address, 2-20
Symbolic Constants, 1-8
Transfer Size, 2-20
CSL (chip support library)
benefits of, 1-2
naming conventions, 1-5
CSL APIs, Generation of the C files, 2-8
CSL APIs Generation (WATCHDOG TIMER Module)
example of, 2-15
Generation of C Files, 2-17
Header File projectcfg.h, illustration of, 2-17
Header File projectcfg_c.c, illustration of, 2-18
main.c File Using Data Generated by the Configuration Tool, illustration of, 2-19
CSL Benefits, 1-2
CSL Functions, Generic, 1-11
CSL Handles, using with data types, 1-7
CSL Tree
csl header files, 2-8
expanded illustration of, 2-5
illustration of, 2-4
CSL_cfgInit, 2-10, 2-11, 3-7, 3-13, 3-19, 3-21, 3-29, 3-34
CSL_init, 2-30

D

DAT, 6-2
  Functions, 6-2
  overview, 6-2
  Primary Summary, 6-2
DAT Functions
  DAT_close, 6-2, 6-5
  DAT_copy, 6-2, 6-3
  DAT_copy2D, 6-2, 6-3
  DAT_fill, 6-2, 6-4
  DAT_open, 6-2, 6-5
  DAT_wait, 6-2, 6-4
DAT_close, 6-2
DAT_close(), 6-2
DAT_copy, 6-2
DAT_copy2D, 6-2
DAT_fill, 6-2
DAT_open, 6-2
DAT_open(), 6-2
DAT_wait, 6-2
DAT_wait(), 6-2
Data Types
  Bool, 1-6
  CSL, 1-6
  DMA_Adr_Ptr, 1-6
  Int16, 1-6
  Int32, 1-6
  PER_Handle, 1-6
  Uchar, 1-6
  Uint16, 1-6
  Uint32, 1-6
  Using CSL Handles, 1-7
declaration list, variables handle and configuration names, 2-8
Declaration section, description of, 2-9
Delete/Rename Options, illustration of, 2-7
Destination address, 2-20
Device Support
  Devices, 1-4
  Device Support Symbols, 1-4
  Large–Model Library, 1-4
  Small–Model Library, 1-4
Directory Structure, 2-24
  Documentation, 2-24
  Examples, 2-24
  Include files, 2-24
  Libraries, 2-24
  Source Library, 2-24
DMA
  Configuration Structure, 7-3
  Functions, 7-3
  Macros, 7-3
  overview, 7-2
  Primary Summary, 7-3
DMA Configuration Manager, 3-8
  Configuring the Object Properties, 3-9
  Creating/Inserting a configuration, 3-8
  Deleting/Renaming an Object, 3-9
  description of, 3-8
  Selecting the Address Formats, 3-10
DMA Configuration Structure, DMA_Config, 7-3, 7-4
DMA Functions
  DMA_close, 7-3, 7-10
  DMA_config, 7-3, 7-6
  DMA_configArgs, 7-3, 7-7
  DMA_getEventId, 7-3, 7-9
  DMA_open, 7-3, 7-9
  DMA_reset, 7-3, 7-10
DMA Header File, example of, 3-12
DMA Initialization
  examples of, 7-13
  using DMA_config(), 7-13
DMA Macros
  DMA_ADDR, 7-11
  DMA_ADDR_H, 7-12
  DMA_FGET, 7-11
  DMA_FGET_H, 7-12
  DMA_FMK, 7-11
  DMA_FSET, 7-11
  DMA_FSET_H, 7-12
  DMA_REG_RMK, 7-11
  DMA_RGET, 7-11
  DMA_RGET_GET, 7-3
  DMA_RGET_H, 7-12
  DMA_RSET, 7-11
  DMA_RSET_H, 7-12
  DMA_RSET_SET, 7-3
  DMA_XXXX_RMK, 7-3
  Using Handle, 7-12
  Using Port Number, 7-11
DMA Module, 3-8
  C Code Generation for, 3-12
  Configuration Manager, 3-8
  overview, 3-8
  Resource Manager, 3-10

DMA Properties Page
illustration of, 3-10
With Handle Object Accessible, , illustration of, 3-12
DMA Resource Manager, 3-10
description of, 3-8
Predefined Objects, 3-11
Properties Page, 3-11
DMA Resource Manager Menu, illustration of, 3-11
DMA Sections Menu, illustration of, 3-8
DMA Source File, example of, 3-13
DMA_AdrPtr, description of, 1-6
DMA_close, 7-2
DMA_config, 3-13, 7-2
DMA_config()
Initializing a, DMA Channel with, 2-20
using, 2-20
DMA_configArgs, 7-2
using, 2-22
DMA_configArgs(), Initializing a DMA Channel with, 2-22
DMA_open, 7-2
DMA_reset, 7-2
DMA0, 3-11
DMA1, 3-11
DMA2, 3-11
DMA3, 3-11
DMA4, 3-11
DMA5, 3-11
Documentation, see also, Directory Structure, 2-24
documentation, related documents from Texas Instruments, v
DOS command line, using, See also, Compiling and Linking with CSL, 2-24
DSP platform, configuration tools, 2-4
DSP/BIOS, 3-1
DSP/BIOS Configuration Tool
CACHE, 2-4
Creating a configuration, 2-12
DMA, 2-4
EMIF, 2-4
GPIO, 2-4
MCBSP, 2-4
PLL, 2-4
TIMER, 2-4

E

EMIF
Configuration Structure, 8-3
Macros, 8-3
overview, 8-2
Primary Summary, 8-3
EMIF Configuration Manager, 3-14
Configuring the Object Properties, 3-15
Creating/Inserting a Configuration Object, 3-14
Deleting/Renaming an Object, 3-15
description of, 3-14
EMIF Configuration Structure, EMIF_Config, 8-3, 8-5
EMIF Functions
EMIF_config, 8-3, 8-7
EMIF_configArgs, 8-3, 8-8
EMIF Header File, 3-18
EMIF Macros
EMIF_ADDR, 8-10
EMIF_FGET, 8-10
EMIF_FMK, 8-10
EMIF_FSET, 8-10
EMIF_REG_RMK, 8-10
EMIF_RGET, 8-10
EMIF_RGET_GET, 8-3
EMIF_RSET, 8-10
EMIF_RSET_SET, 8-3
EMIF_XXXX_RMK, 8-3
Using Port Number, 8-10
EMIF Module, 3-14
overview, 3-14
EMIF Properties Page, 3-16
EMIF Resource Manager, 3-16
descriptio of, 3-14
Properties Page, 3-16
EMIF Resource Manager Dialog Box, 3-17
EMIF Sections Menu, illustration of, 3-14
EMIF Source File (Body Section), 3-19
EMIF Source File (Declaration Section), 3-18
EMIF_config, 3-19, 8-2
EMIF_configArgs, 8-2
EMIF_open, 3-19
Examples
MCBSP, 11-14
see also, Directory Structure, 2-24

F

Far Calls, 2-27
FIELD, 1-8
  explanation of, 1-9
fieldval, explanation of, 1-9
funcArg, 1-5
Function, 1-5
Function Argument, 1-5
Function Inlining, using, 2-31
Functions
  CHIP, 5-3
  CSL, 1-11
  DAT, 6-3
  DMA, 7-6
  EMIF, 8-7
  IRQ, 10-8
  PWR, 13-3
  TIMER, 14-5

G

Generation of the C Files, 2-8
  Header files, 2-8
  Source files, 2-8
Getting Started
  Modification of C code, 2-14
  Modification of the Project folder, 2-12
GPIO Configuration Manager, 3-20
  description of, 3-20
GPIO Macros
  MCBSP_ADDR, 9-3
  MCBSP_FGET, 9-3
  MCBSP_FMK, 9-3
  MCBSP_FSET, 9-3
  MCBSP_REG_RMK, 9-3
  MCBSP_RGET, 9-3
  MCBSP_RSET, 9-3
  Using Port Number, 9-3
GPIO Module, overview, 3-20
GPIO Properties Page, illustration of, 3-21
GPIO Source File (Body Section), example of, 3-22
GPIO_RSET(), 3-22

H

Header file, Projectcfg.h, 2-8
Header File projectcfg.h, 2-8
  WATCHDOG TIMER Module, illustration of, 2-17
How To Use CSL, overview, 2-3

I

Include Files, see also, Directory Structure, 2-24
Include section, description of, 2-9
Initializing Registers, 1-12
Inserting a Configuration Object, illustration of, 2-6
Int16, description of, 1-6
Int32, description of, 1-6
IRQ
  Configuration Structure, 10-2
  Functions, 10-3
  overview, 10-2
IRQ Configuration Structure, IRQ_Config, 10-2, 10-7
IRQ Functions
  IRQ_clear, 10-3, 10-8
  IRQ_config, 10-3, 10-8
  IRQ_configArgs, 10-3, 10-9
  IRQ_disable, 10-3, 10-9
  IRQ_enable, 10-3, 10-10
  IRQ_getArg, 10-3, 10-10
  IRQ_getConfig, 10-3, 10-10
  IRQ_globalDisable, 10-3, 10-11
  IRQ_globalEnable, 10-3, 10-11
  IRQ_globalRestore, 10-3, 10-12
  IRQ_map, 10-3, 10-12
  IRQ_plug, 10-3, 10-13
  IRQ_setArg, 10-3, 10-13
  IRQ_setVecs, 10-3, 10-14
  IRQ_test, 10-3, 10-14
IRQ_EVT_NNNN, 10-4
  Event List, 10-4
IRQ_EVT_WDTINT, 10-6

L

Libraries, see also, Directory Structure, 2-24
Linker Command File
  creating, 2-30
  using, 2-30
Linking, with CSL, 2-24

M

Macro, 1-5

Macros
- CACHE, 4-8
- CSL, 1-9
- DMA, 7-11
- EMIF, 8-10
- Generic, 1-9
- Generic, handle–based, 1-10
- GIO, 9-3
- MCBSP, 11-12
- PLL, 12-8
- PWR, 13-4
- TIMER, 14-9

Macros, generic description of
- CSL, 1-9
- FIELD, 1-9
- fieldval, 1-9
- PER, 1-9
- REG#, 1-9
- regval, 1-9

main.c File Using Data Generated by the Configuration Tool, WATCHDOG TIMER Module, illustration of, 2-19

MCBSP
- Configuration Manager, 3-23
- Example, 11-14
- Functions, 11-3
- Macros, 11-3
- Configuration Structure, 11-3
- Primary Summary, 11-3

MCBSP Configuration Manager, 3-23
- Configuring the Object Properties, 3-24
- Creating/Inserting a Configuration Object, 3-23
- Deleting/Renaming an Object, 3-24

MCBSP Configuration Structure, MCBSP_Config, 11-3, 11-4

MCBSP Functions
- MCBSP_close, 11-3, 11-10
- MCBSP_config, 11-3, 11-6
- MCBSP_configArgs, 11-3, 11-7
- MCBSP_getRcvEventID, 11-3, 11-9
- MCBSP_getXmtEventID, 11-3, 11-9
- MCBSP_open, 11-3, 11-10
- MCBSP_reset, 11-3, 11-11
- MCBSP_start, 11-3, 11-11

MCBSP Header File, Example of, 3-27

MCBSP Macros
- Using Handle, 11-13
- MCBSP_ADDR, 11-12
- MCBSP_ADDR_H, 11-13
- MCBSP_FGET, 11-12
- MCBSP_FGET_H, 11-13
- MCBSP_FMK, 11-12
- MCBSP_FSET, 11-12
- MCBSP_FSET_H, 11-13
- MCBSP_REG_RMK, 11-12
- MCBSP_RGET, 11-12
- MCBSP_RGET_GET, 11-3
- MCBSP_RGET_H, 11-13
- MCBSP_RSET, 11-12
- MCBSP_RSET_H, 11-13
- MCBSP_RSET_SET, 11-3
- MCBSP_XXXX_RMK, 11-3
- Using Port Number, 11-12

MCBSP Module, 3-3, 3-23
- C Code Generation for, 3-27
- overview, 3-23

MCBSP Properties Page
- illustration of, 3-25
- With Handle Object Accessible, , illustration of, 3-27

MCBSP Resource Manager, 3-23, 3-26
- description of, 3-23
- Predefined Objects, 3-26
- Properties Page, 3-26

MCBSP Resource Manager Menu, illustration of, 3-26

MCBSP Sections Menu, illustration of, 3-23

MCBSP Source File (Body Section), example of, 3-29

MCBSP Source File (Declaration Section), example of, 3-28

MCBSP_close, 11-2

MCBSP_config, 3-29, 11-2

MCBSP_configArgs, 11-2

MCBSP_open, 3-29, 11-2

MCBSP_reset, 11-2

MCBSP0, 3-26

MCBSP1, 3-26

MCBSP2, 3-26

mcbspCfg, 3-23

memberName, 1-5

Modifying the C File, example of, 2-14

Module
    DMA, 3-8
    GPIO, 3-20
    PLL, 3-30
    TIMER, 3-35

N

notational conventions, iv

O

Object Types, 1-5

P

PER, 1-8
    explanation of, 1-9
PER_ADDR, 1-10
PER_ADDR_H, 1-10
PER_close, 1-12
PER_Config
    example of, 1-13
    explanation of, 1-12
PER_config, 1-11
PER_ConfigArgs
    example of, 1-13
    explanation of, 1-12
PER_configArgs, 1-11
PER_FGET, 1-10
PER_FGET_H, 1-10
PER_FMK, 1-9
PER_FSET, 1-10
PER_FSET_H, 1-10
PER_funcName(), 1-5
PER_Handle, description of, 1-6
PER_MACRO_NAME, 1-5
PER_open, 1-11
PER_REG_DEFAULT, 1-8
PER_REG_FIELD_DEFAULT, 1-8
PER_REG_FIELD_SYMVAL, 1-8
PER_REG_RMK, 1-9
PER_reset, 1-12
PER_RGET, 1-9
PER_RGET_H, 1-10
PER_RSET, 1-9
PER_RSET_H, 1-10
PER_start, 1-12
PER_Typename, 1-5
PER_varName(), 1-5
PERIPHERAL Configuration Manager, description of, 2-6
Peripheral Modules, 1-3
    CACHE, 1-3
    CHIP, 1-3
    DAT, 1-3
    Description of, 1-3
    DMA, 1-3
    EMIF, 1-3
    GPIO, 1-3
    Include Files, 1-3
    IRQ, 1-3
    MCBSP, 1-3
    Module Support Symbols, 1-3
    PLL, 1-3
    PWR, 1-3
    TIMER, 1-3
PERIPHERAL Resource Manager, description of, 2-6
Peripheral_config, 2-10
Peripheral_open, 2-10
PLL
    Configuration Structure, 12-3
    Functions, 12-3
    Macros, 12-3
    Primary Summary, 12-3
PLL Configuration Manager, 3-30
    Configuring the Object Properties, 3-31
    Creating/Inserting a configuration, 3-30
    Deleting/Renaming an object, 3-31
    description of, 3-30
PLL Configuration Structure, PLL_Config, 12-3, 12-4
PLL Functions
    PLL_close, 12-3, 12-5
    PLL_config, 12-3, 12-6
    PLL_configArgs, 12-3, 12-6
    PLL_open, 12-3, 12-5
PLL Header File, example of, 3-33
PLL Macros
    PLL_FGET, 12-8
    PLL_FMK, 12-8
    PLL_FSET, 12-8
    PLL_REG_RMK, 12-8
    PLL_RGET, 12-8

PLL_RGET_SET, 12-3
PLL_RSET, 12-8
PLL_RSET_SET, 12-3
PLL_XXXX_RMK, 12-3
Using Port Number, 12-8

PLL Module, overview, 3-30

PLL Properties Page, illustration of, 3-31

PLL Resource Manager, 3-32
description of, 3-30
Properties page, 3-32

PLL Resource Manager Menu, illustration of, 3-32

PLL Sections Menu, illustration of, 3-20, 3-30

PLL Source File (Body Section), example of, 3-34

PLL Source File (Declaration Section), example of, 3-34

PLL_close, 12-2

PLL_config, 3-34, 12-2

PLL_configArgs, 12-2

PLL_open, 12-2

Practice Summary, illustration of, 2-13

predefined handle and configuration objects, accessing, 2-8

Predefined Objects, 3-11, 3-26
DMA0, 3-11
DMA1, 3-11
DMA2, 3-11
DMA3, 3-11
DMA4, 3-11
DMA5, 3-11
MCBSP0, 3-26
MCBSP2, 3-26
TIMER, 3-38
TIMER0, 3-38
TIMER1, 3-38

project.cdb, 2-8

projectcfg.h, 2-8

projectcfg_c.c, 2-8

Properties Page, 3-11
TIMER, 3-38

Properties Page Options, example of, 2-9

Properties Pages
Advanced A and B Pages, 3-9
Autoinit, 3-9
Source/Destination, 3-9
Transfer Modes, 3-9

Properties Pages of the Non–Multiplexed GPIO Configuration, GPIO, 3-20

PWR
Functions, 13-2
overview, 13-2

PWR Functions, PWR_powerDown, 13-2

PWR Macros
PWR_ADDR, 13-4
PWR_FGET, 13-4
PWR_FMK, 13-4
PWR_FSET, 13-4
PWR_REG_RMK, 13-4
PWR_RGET, 13-4
PWR_RSET, 13-4

R

REG, 1-8

REG#, explanation of, 1-9

Registers
initializing, 1-12
PER_Config, 1-13
PER_ConfiArgs, 1-13

regval, explanation of, 1-9

related documents from Texas Instruments, v

Resource Management, 1-6

Resource Manager
CACHE, 3-5
MCBSP, 3-26
TIMER, 3-37

Resource Manager Properties Page, illustration of, 2-10

S

Show Dependency Option, illustration of, 2-7

Source address, 2-20

Source File, 3-28

Source file, Projectcfg_c.c, 2-8

Source File projectcfg_c.c, 2-9
Body section , 2-10
Declaration section, 2-9
Include section, 2-9
WATCHDOG TIMER Module, illustration of, 2-18

Source Library, see also, Directory Structure, 2-24

Source/Destination, 3-9

static inline, 2-31

Structure Member, 1-5
Symbolic Constant Values, 1-8
Symbolic Constants, Generic, 1-8
SYMVAL, 1-8

T

TIMER
  C Code Generation for, 3-39
  Configuration Structure, 14-3
  Functions, 14-3
  Macros, 14-3
  Primary Summary, 14-3
TIMER Configuration Manager, 3-35
  Configuring the Object Properties, 3-36
  Creating/Inserting a configuration, 3-35
  Deleting/Renaming an Object, 3-36
  description of, 3-35
TIMER Configuration Structure
  TIMER_close, 14-3
  TIMER_Config, 14-3, 14-4
  TIMER_reset, 14-3
  TIMER_tintoutCfg, 14-3
TIMER Functions
  TIMER_close, 14-7
  TIMER_Config, 14-5
  TIMER_config, 14-3
  TIMER_configArgs, 14-3, 14-5
  TIMER_getEventID, 14-6
  TIMER_open, 14-3, 14-6
  TIMER_reset, 14-7
  TIMER_start, 14-3, 14-7
  TIMER_stop, 14-3, 14-8
  TIMER_tintoutCfg, 14-8
Timer Header File, example of, 3-39
TIMER Macros
  TIMER_ADDR, 14-9
  TIMER_ADDR_H, 14-10
  TIMER_FGET, 14-9
  TIMER_FGET_H, 14-10
  TIMER_FMK, 14-9
  TIMER_FSET, 14-9
  TIMER_FSET_H, 14-10
  TIMER_REG_RMK, 14-9
  TIMER_RGET, 14-3, 14-9
  TIMER_RGET_H, 14-10
  TIMER_RSET, 14-3, 14-9
  TIMER_RSET_H, 14-10
  TIMER_XXXX_RMK, 14-3
  Using Handle, 14-10
  Using Port Number, 14-9
TIMER Module, 3-35
  overview, 3-35
TIMER Properties Page, illustration of, 3-37
Timer Properties Page With Handle Object Accessible, illustration of, 3-39
TIMER Resource Manager, 3-37
  description of, 3-35
Timer Resource Manager Menu, illustration of, 3-37
Timer Sections Menu, illustration of, 3-35
Timer Source File (Body Section), example of, 3-40
Timer Source File (Declaration Section), example of, 3-40
TIMER0, 3-38
TIMER1, 3-38
trademarks, vi
Transfer Modes, 3-9
Transfer size, 2-20
Typedef, 1-5

U

Uchar, description of, 1-6
Uint16, description of, 1-6
Uint32, description of, 1-6

V

Variable, 1-5

W

WATCHDOG TIMER1 Device
  configuration of, 2-15
  configuration of, illustration, 2-16